THE SECRET GUIDE TO COMPUTERS

Volume 2

Deep Secrets

This book digs deeper into the mysteries of computers
than any book ever published before.

It includes complete courses in
the microcomputer industry, business applications,
advanced programming in BASIC, FORTRAN, PASCAL, COBOL,
20 other high-level languages, 9 assembly languages,
numerical analysis, computer-literacy curricula,
how to get a job, and more!

Praised by all the famous microcomputer magazines,
The Secret Guide to Computers has changed thousands of lives.
Let it change yours.

Eleventh edition © 1984 by Russ Walter, $14. To order, use coupon on back page.

Digitized by the Internet Archive
in 2025

https://archive.org/details/secretguidetocom0002russ

Dear You!

This volume contains a greater quantity of useful information about computers than any other book ever published. I hope you like it.

It explains the deepest secrets about microcomputers, business applications, advanced programming in BASIC, FORTRAN, PASCAL, COBOL, 22 high-level programming languages, 9 assembly languages, numerical analysis, and computer careers. This eleventh edition includes the latest news, views, tricks, and shticks.

Whenever you have a question about computers—or about life—phone me at 617-266-8128, and I'll help you, free, even if your question is weird or personal. You can call 24 hours: I'm almost always in, and I sleep only lightly.

I'll help you decide which computer and software to buy, help you write programs, debug them, cheer you up, or whatever else you wish. No charge!

To get the best service from me when you phone, begin by telling me your name, the country or state you're calling from, which volumes of the Guide you own (you probably own "volumes 1 and 2 of the eleventh edition"), and a one-sentence summary of your question. Then we'll chat about your question's details. Remember: to get the free help, phone me; do *not* write!

In your hands, you have volume 2 of *The Secret Guide to Computers*. To thoroughly understand it, read volume 1 first, which you can order by using the coupon on the back page.

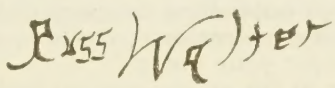
The Secret Guide to Computers is the only set of books ever published that's been extensively praised by *all* the popular microcomputer magazines. I thank my current staff (Priscilla Grogan, Irene Vassos, Gloria Pawlyk, and Janet McSwiggin), their predecessors, my hundreds of associates, and the thousands of readers (like you) who have contributed suggestions that made this book so wonderful. This month's winner of the Helpful Reader Award goes to David Rowe of Augusta, Maine, for his long letter that contained many useful suggestions.

I've been revising the Guide for 14 years. The current two-volume set contains 750 pages altogether. Here's a quick history: edition 0 (copyright 1972) had 17 pages; edition 1 (1972) 12 pages; edition 2 (1972) 20 pages; edition 3 (1972) 32 pages; edition 4 (1973) 63 pages; edition 5 (1973) 73 pages; edition 6 (1974) 260 pages; editions 7 & 8 & 9 (1976-1979) 410 pages; edition 10 (1980-1982) 696 pages; edition 11 (1984) 750 pages.

For edition 12, I'm writing an expanded analysis of Apple's Macintosh computer, mouse & joystick commands, sequential-access & random-access files, PEEK & POKE statements, LOGO, FORTH, DBASE 2, Lotus 1-2-3, and 16-bit assembly languages. I'm also reorganizing the material, so that some of the information that's at the beginning of volume 2 will be integrated into volume 1. I'm also updating the passages that are obsolescent, and correcting the typographical errors that my computerized proofreader didn't find. If you have any additional suggestions for edition 12, please tell me.

To get copies of The Secret Guide to Computers for your friends at a discount, plus free info about edition 12, the children's edition, computer courses, and The Secret Guide to Tricky Living, use the coupon on the back page.

At your service,
Your computer butler,



PRAISED BY MAGAZINES

The Secret Guide to Computers has been praised by all the famous microcomputer magazines. . . .

POPULAR COMPUTING MAGAZINE says, "Russ Walter is king of the East Coast computer cogoscenti. His Secret Guide is the biggest bargain in computer tutorials in our hemisphere. If CBS ever decides to replace Andy Rooney with a '60 Minutes' computer pundit, they'd need to look no further than Russ Walter. The Secret Guide is the first collection of computer writings that one might dare call literature."

INTERFACE AGE MAGAZINE says, "The Secret Guide claims to cover all the facets of computers simply but thoroughly, and to be foolproof, carefully tested and revised, fun, simple to read, requiring no math background, and popular among kids and adults who want straightforward talk. The book meets all those claims. How did Russ Walter do it?"

PERSONAL COMPUTING MAGAZINE says, "Russ Walter's approach to text-writing sets a new style that other authors might do well to follow. If more college texts were written in the Russ Walter style, more college students would reach their commencement day."

SOFTALK says, "Many leaders of the computer industry are alumni of Russ Walter's courses. The Guide has high energy, humor, and flair. Russ pulls no punches, firing well-deserved salvos at many sacred cows. The Guide has long been a cult hit."

THE WHOLE EARTH CATALOG, in its Coevolution Quarterly update, says, "The Secret Guide is one of the few introductory books to be fiercely honest, and the only introductory survey of equipment that's kept up to date."

CIDER PRESS says, "The Guide is as difficult to lay aside as a first-rate mystery. It should be given to all beginners with the purchase of their computers."

INFOWORLD says, "Russ Walter is respected in many parts of the country. People call him at all hours of the night from as far away as Australia for advice."

COMPUTER UPDATE says, "Russ Walter is Boston's computer guru. He's years ahead of the pack that claims to have ways of instructing computer novices."

CREATIVE COMPUTING says, "The Guide is fascinating, easy to understand, an excellent book at a ridiculously low price."

CLASSROOM COMPUTER NEWS says, "Russ's courses are intensive and inexpensive."

ESQUIRE says, "The Guide contains lots of fact and opinion untainted by bias."

MICROCOMPUTING MAGAZINE says, "Plan ahead: get in on the secret now."

COMPUTE MAGAZINE says, "Russ Walter is an industry leader."

BYTE MAGAZINE says, "Get Russ Walter's advice."

FAN MAIL

From our readers, we've received *thousands* of letters and phone calls, praising us. For example, here are the first four we received. . . .

Head of a meat factory: "I spent $100 on books. Yours was the only one I could understand."

Head of a computer center: "Terrific! Your book is the best. You have the others beat by a mile."

Head of a large publishing company: "Although I find the machines deplorable, you've taken such a refreshing, comical, and interesting approach that even the likes of me can enjoy your books."

Head engineer at the atomic-weapon design headquarters: "The Secret Guide has assumed the importance here of a very critical top-secret document. It will be required reading for the whole crew."

CONTENTS OF THIS VOLUME

Volume 2 consists of ten chapters.

Chapter 1 (Microcomputers) analyzes each of the popular microcomputer companies. It shows why each company entered the microcomputer marketplace, and what mistakes they made. It lambasts the three great marketing myths: the myth of CP/M, the myth of MS-DOS, and the myth of computer power. You'll find out more about Apple, Commodore, Radio Shack, IBM, Coleco, Atari, and their competitors than you ever knew before. To find out which computer's the easiest to program, peek at the comparison chart, which will surprise you. You'll also giggle at cars that talk, computers that do the bossanova, and computerized Scrabble.

Chapter 2 (Business Applications) gives you candid advice about the hassles of computerizing your business. It also looks deeply and critically at word processing and data-management systems.

Chapter 3 (Advanced Programming using BASIC) covers everything important about BASIC that I was afraid to mention in volume 1. It contains the results of the newest research about programming: it reveals the best way to alphabetize (never before published!) and the best way to make the computer shuffle a deck of cards (never before published!). It explains how to program your own data-management system, by using random-access data files and trees. It explains how to make the Atari do fancy graphics and music, and how to make the Epson printer create fancy fonts. You'll learn the tricks that work on all advanced microcomputers, such as how to *trap* an error, how to *format* your output, how to streamline your logic by saying "ELSE" and "ON", and how to make BASIC run faster. Those tricks, which many programmers spend *years* trying to discover, are all revealed for you here!

Chapters 4, 5, 6, and 7 cover a wide variety of computer languages; each chapter is the equivalent of a college course. Chapter 4 is a one-semester course in FORTRAN. Chapter 5 is a half-semester course in PASCAL. Chapter 6 is a one-semester course in COBOL. Chapter 7 is a one-semester course in comparative languages: it delves into ADA, ALGOL, APL, APT, BASIC, C, COBOL, DBASE 2, DYNAMO, EASY, FORTH, FORTRAN, GPSS, LISP, LOGO, PASCAL, PILOT, PL/I, RPG, SNOBOL, SPSS, and VISICALC.

Chapter 8 (Bit Fiddling) dissects the computer's innermost workings. After explaining number systems (binary, octal, and hexadecimal) and character codes (ASCII and EBCDIC), it tutors you in assembly language, by exposing my SEXY ASS (which is the newest assembly language). Then it tutors you in machine language, by forcing you to work with the SEXY MACHO (which is the newest machine language). After all that, you zoom through the most popular assemblers and machine languages for microprocessor families (6502, Z-80, 8080/8085/8086/8088, and 6800/6809/68000) and for larger computers (by IBM, DEC, and CDC). Along the way, you'll also compare the operating systems for maxicomputers, minicomputers, and microcomputers.

Chapter 9 (Abstract Applications) shows how to make the computer draw 3-dimensional pictures. It also includes a one-semester course in numerical analysis.

Chapter 10 (Changing Lives) shows how the computer can improve your life and the lives of others—or wreck them! It explains the newest tricks for getting a computer job. It critiques educational software and computer-literacy curricula. It advises how to increase your own knowledge about computers, so you can turn yourself into a computer guru. It tells you the best computer books and magazines ever published by other publishers.

CHAPTER 1: MICROCOMPUTERS

MICRO HISTORY *(let's look back, maxicomputers, minicomputers, S-100 bus computers, computers with built-in keyboards, improvements, Texas Instruments & Atari, Sinclair & Osborne & backlash, IBM PC, recent developments, today's marketplace): page 8*
MYTHS *(CP/M, MS-DOS, power): page 15*
APPLE *(slots, Apple's reputation): page 17*
COMMODORE *(why Commodore, how Commodore began, dealing with competitors, how Commodore launched its PET, problem with RAM, problem with tape, success in schools, success in Europe, VIC, Commodore 64, Commodore today): page 18*
RADIO SHACK *(how the TRS-80 began, the Millers, dealing with the public, Radio Shack today, Pocket Computers, Color Computers, Model 100, disk-based computers): page 26*
IBM *(PC Junior, PC Senior): page 30*
COLECO *(what's Coleco, the Adam announcement, the price war, disappointments, Adam's manuals, phone calls, dispute, conclusion, for Adam owners): page 32*
ATARI *(why Atari, how Atari began, Atari 400 & 800, new products): page 40*
PROGRAMMABILITY *(Dartmouth BASIC, DEC BASIC, DEC's competitors, Microsoft, pay your Bill, comparison chart): page 44*
STRANGE DEVICES *(the talking car, Casio synthesizer, game machines): page 48*

CHAPTER 2: BUSINESS APPLICATIONS

GROUNDWORK *(how to buy a computer, repairs, manuals, programs, data entry, coworkers, summary, minimum req., price trends, kinds of prices): page 52*
WORD PROCESSING *(what is word processing, how a high-quality word processor works): page 56*
SUPER SCRIPSIT *(what's Super Scripsit, how to move the cursor, etc.): page 57*
DATA MANAGEMENT *(what's a data-management system, file-cabinet jargon, a sample file, data management versus word processing): page 62*
PFS *(what's PFS, four disks, how to get started, how to copy, how to search, advanced features, limitations): page 64*

CHAPTER 3: ADVANCED PROGRAMMING USING BASIC

STRINGY THINGS *(extracting from strings, changing the middle, adding strings, searching in a string, clock, string-number conversion, ASCII, repeating characters): page 70*
PRINT USING *(fundamentals, multiple numbers, large numbers, final semicolon, advanced formats): page 75*
SWAPPING *(how to swap, shuffling, alphabetizing): page 78*
ADVANCED MATH *(trigonometry, EXP, logarithms, sign): page 85*
ATARI BASIC *(differences, how the computer gripes, music, graphics): page 88*
TRICKY TYPING *(two SHIFT keys, LOCK key, numeric keypad, canceling a line, reversing the color, blinking, fat characters): page 94*
LOVABLE LOGIC *(how to say "IF", ON . . . GO TO, ON . . . GOSUB, STOP versus END, error traps, how to speed up the computer): page 96*
DATA FILES *(sequential access, random access): page 102*
CREATE A DATA BANK *(chronological order, alphabetical order, trees, disks, jargon): page 106*

CHAPTER 4: FORTRAN

FUN *(what's FORTRAN, simple programs, carriage controls, popular formats, GO TO, STOP): page 120*
MATH *(numbers, operations, advice about variables): page 124*
I/O *(READ, omitting formats): page 130*
PRETTIER PROGRAMS *(comments, continuation): page 133*
LOGIC *(IF, computed GO TO, DO): page 134*
LISTS *(subscripts, implied DO, DATA, string variables): page 137*
FANCY FORMATS *(double apostrophe, T format, blank records, repeated formats, E format, P format): page 142*
FUNCTIONS *(square root, FLOAT, random numbers, maxima & minima, absolute value, remainder, trigonometry, calculus): page 144*
EXOTIC FEATURES *(DOUBLE PRECISION, COMPLEX, subroutines, your own functions, files): page 148*

CHAPTER 5: PASCAL
FUN *(what's PASCAL, simple programs, multiple writing, math, variables, READ, IF): page 160*
LOOPS *(FOR, REPEAT . . . UNTIL, WHILE): page 166*
EXOTIC FEATURES *(alternatives, comments, GOTO, PROCEDURE, CHAR, subscripts, writing strings, reading strings): page 170*

CHAPTER 6: COBOL
FUN *(what's COBOL, simple, abridgments, fancy displays, PERFORM): page 180*
VARIABLES *(string variables, numeric variables, decimals, negatives, ACCEPT, editing): page 188*
LOGIC *(IF, UNTIL, GO TO, THRU): page 196*
STRUCTURES *(files, group items, SORT, MERGE, subscripts): page 202*
EXOTIC FEATURES *(IDENTIFICATION DIVISION, comments, DECIMAL-POINT IS COMMA): page 216*

CHAPTER 7: ALTERNATIVE LANGUAGES
LANGUAGE SURVEY *(eight categories, family tree, etc.): page 220*
CLASSICAL LANGUAGES *(APT, FORTRAN, ALGOL, DYNAMO, RPG, LISP, COBOL, GPSS, SNOBOL, BASIC, APL, PL/I, SPSS): page 226*
MODERN LANGUAGES *(LOGO, FORTH, PILOT, PASCAL, C, VISICALC, ADA, DBASE 2, EASY): page 270*

CHAPTER 8: BIT FIDDLING
NUMBER SYSTEMS *(computers aren't human, binary, octal, hexadecimal): page 282*
CHARACTER CODES *(ASCII, EBCDIC, bytes, nibbles): page 286*
ASSEMBLY LANGUAGE *(SEXY ASS, electronic boxes, how to copy a byte, arithmetic, immediate addressing, video RAM, labels, machine language, system software): page 288*
FLAGS *(carry flag, negatives, flag register, conditional jumps): page 294*
SCHEDULING THE USERS *(CPU, memory, I/O devices): page 297*
PERSONALITY *(computer persona, boss, conversationalist, drudge): page 301*
OPERATING SYSTEMS *(IBM, DEC, Apple, CP/M): page 302*
HOW THE CPU THINKS *(words, character codes, BCD, kinds of languages, electronic boxes, program counter, arithmetic/logic unit, flags, speed, parity, UAL, addressing modes): page 304*
THE CPU'S MAP *(nine maps, PDP-8, PDP-10, PDP-11, CDC, IBM 360, microcomputers): page 316*
INSTRUCTION SETS *(PDP-10, PDP-8, PDP-11, CDC, IBM 360, micro): page 319*

CHAPTER 9: ABSTRACT APPLICATIONS
3-D DRAWING *(coordinates, drawing an object, computerizing): page 340*
NUMERICAL ANALYSIS *(twin problems that confuse the computer, how to reduce round-off error, how to estimate an unknown answer, how to solve an equation, how to solve simultaneous equations, how to compute the area under a curve): page 342*

CHAPTER 10: CHANGING LIVES
YOUR CAREER *(how to become an expert, get a job, what to charge): page 354*
EDUCATION *(curriculum, school applications, purchasing, management): page 358*
ANCIENT HISTORY *(before computers, Charles Babbage, Lady Lovelace, Herman Hollerith, World War II, the first generation, the second generation, the dawn of the third generation, the era of boredom, the era of personal computing): page 364*
SPELLING *(errors, choices): page 373*
IN THE HOME *(dreamers, work at home, electronic shopping, appliances, government, newspapers, medicine, the whole family, Compuserve): page 374*
DANGERS *(change society, errors, unemployment, quantification, asocial behavior, irresponsibility, concentrated power, crime, invaded privacy): page 380*
JOBS *(programmer, management, sales, original jobs): page 384*
WHAT TO READ *(periodicals, where to get books, best books, books for beginners, books about alternative computer languages, books about operating systems, books about data-base management systems, books about advanced programming techniques, books about assembly languages): page 392*
GET PERSONAL *(phone me, blitz courses): page 396*
COUPON*: page 398*

ご拡大下さい！

MICRO COMPUTERS

MICRO HISTORY

LET'S LOOK BACK

How did the computer industry develop? Let's look back. . . .

MAXICOMPUTERS

The first programmable computers were invented in the 1940's, because of Word War II. Most of them were invented at universities (such as Harvard, the University of Pennsylvania, and M.I.T.), with funds from the War Department (which today is more delicately called the "Defense Department"). Each of those computers was, as they say in the art world, a "signed original". No two of them were alike.

The first computer to be mass-produced was the Univac 1, in 1951. It was a large and expensive maxicomputer. Since chips hadn't been invented yet, its RAM consisted of *mercury delay lines* instead, and therefore held just 9K. Forty-six of those computers were built. The manufacturer was Remington Rand (which later became Sperry-Rand).

In 1953, a typewriter company named "IBM" entered the computer marketplace and began selling computers that competed against the Univac 1. Since IBM's salesmen were slicker than Remington Rand's, IBM soon became the dominant computer manufacturer, and has remained so ever since.

MINICOMPUTERS

The first popular *mini*computer was shipped in 1965. It was called the *PDP-8* and was manufactured by Digital Equipment Corporation (DEC).

In 1970, DEC began shipping a better minicomputer, called the *PDP-11*. It became the most popular minicomputer in the world. Later, DEC has been selling a souped-up version, called the *VAX*.

The typical small business bought a minicomputer. The typical *large* business bought a *maxi*computer, but another approach was possible: instead of buying a maxicomputer, the business could buy several minicomputers, give each department its own minicomputer, and run wires between the minicomputers to let them communicate with each other. The idea of giving each department its own minicomputer (instead of making everybody share a maxicomputer) was called *distributed processing* and offered two advantages:

If a computer broke, only one department would suffer, instead of the whole company.

No department could hog the entire computer system and screw all the other departments.

S-100 BUS COMPUTERS

In 1975, the first popular *micro*computer was shipped. It was called the *Altair* and was manufactured by a company called *MITS*. It cost just $395.

It was just a box that contained a CPU and very little RAM: only ¼ of a K! It included no printer, no disk, no tape, no ROM, no screen, and not even a keyboard! The only way to communicate with the computer was to throw 25 switches and watch 36 blinking lights. It didn't understand BASIC or any other high-level computer language: to learn how to throw the switches and watch the blinking lights, you had to take a course in "machine language". You also had to take a course in electronics—because the $395 got you just a kit that you had to assemble yourself, by using a soldering iron and reading electronics diagrams. Moreover, when you finished building the kit, you found that some of the parts were missing or defective, so that you had to contact MITS for new parts.

That computer contained several empty slots, to hold extra PC boards. Eventually, many companies invented PC boards to put into those slots; those PC boards would let you insert extra RAM and also attach terminals, televisions, tape recorders, disk drives, and printers. Those PC boards were expensive.

Bill Gates invented a way to make the Altair handle BASIC. He called his method *Microsoft BASIC*. He patterned it after DEC's BASIC; but he included extra features that exploited the Altair's ability to be "personal", and he eliminated features that would require too much RAM.

Gary Kildall invented a DOS that the Altair could use. He called his DOS *CP/M*.

Many companies built computers that imitated the Altair. Those imitations became more popular than the Altair itself. Eventually, MITS (the company that made the Altair) went out of business.

The computers that imitated the Altair were called *S-100 bus computers*, because they each used a cable that contained 100 wires.

In those days, the microcomputer industry was standardized. Each popular microcomputer used Microsoft BASIC, CP/M, and the S-100 bus. The microcomputer was just a box that contained PC cards; it had no keyboard, no screen, and no disk drive. A cable went from the microcomputer to a terminal (keyboard with screen), which was priced separately. Another cable went from the microcomputer to a disk drive, which was also priced separately.

COMPUTERS WITH BUILT-IN KEYBOARDS

In 1977, four companies began selling microcomputers that had built-in keyboards, so you didn't have to buy a terminal. Their computers became immediately popular. The four companies were Processor Technology, Apple, Commodore, and Radio Shack.

Processor Technology's computer was called the *Sol 20*, in honor of Solomon Libes, who was an editor of Popular Electronics. Apple's computer was called the *Apple 2*, because it was an improvement on the Apple 1, which had lacked a built-in keyboard. Commodore's computer was called the *PET*, because its concept was invented by the same guy who had invented pet rocks. (He figured that if people were stupid enough to buy novelty items called "pet rocks", they'd really go ape over "pet computers".) Radio Shack's computer was called the *TRS-80*, because it was manufactured by Tandy's Radio Shack and contained a Z-80 CPU.

For a fully assembled computer, Processor Technology charged $1850, Apple charged $970, Commodore charged $595 (but quickly raised the price to $795), and Radio Shack charged $599 (but soon lowered the price to $499). Notice that Commodore and Radio Shack had the lowest prices. Also, the low prices from Commodore and Radio Shack *included* a monitor, whereas the prices from Processor Technology and Apple didn't. So Commodore and Radio Shack were the real "bargains".

The popularity of those computers was directly related to their prices. The cheapest computer (Radio Shack's) was the most popular; Commodore was the second most popular; Apple was the third most popular; and Processor Technology, after a brief fling of popularity, went bankrupt.

So at that time, you could buy four popular kinds of microcomputers: the cheapest was Radio Shack's TRS-80; next up was Commodore's PET; next up was the Apple 2; and the most expensive kind of microcomputer was the CP/M S-100 bus system, which was the oldest kind and therefore had accumulated the greatest quantity of business software.

IMPROVEMENTS

In 1978 and 1979, the three main companies (Apple, Commodore, and Radio Shack) improved their computers.

The improved Apple 2 was called the *Apple 2-plus*. The improved Commodore PET was called the *Commodore Business Machine (CBM)*. The improved Radio Shack TRS-80 was called the *TRS-80 model 2*.

After announcing the Apple 2-plus, Apple Computer Company stopped selling the plain Apple 2.

Commodore continued selling its old computer (the PET) to customers who couldn't afford the new version (the CBM), which was more expensive. Likewise, Radio Shack continued selling its model 1 to customers who couldn't afford the model 2.

TEXAS INSTRUMENTS & ATARI

In 1979, Texas Instruments (TI) and Atari entered the microcomputer marketplace and began selling low-priced computers.

TI's microcomputer was called the *TI 99/4*. Atari offered *two* microcomputers, called the *Atari 400* and the *Atari 800*.

TI charged $1150. Atari charged $1000 for the regular model (the Atari 800) and $550 for the stripped-down model (the Atari 400).

TI's price included a color monitor. Atari's prices did *not* include a screen; you were to attach Atari's computers to your home's TV.

TI's computer was terrible, especially its keyboard. The Atari 800 computer was wonderful: it had a wonderfully easy-to-use keyboard, offered a wonderfully easy way to edit your programs, produced wonderfully gorgeous colors on your TV, was wonderfully child-proof (safe for little kids), offered wonderfully dazzling games, and, when you considered everything you were getting, had a wonderfully low price! It was cheaper than an Apple (whose price had by then risen to $1195) and yet was much *better* than an Apple.

From that description, you'd think that the Atari 800 would become the world's best-selling computer, and that the TI 99/4 would become an immediate flop and stop being sold. Indeed, that's what most computer experts hoped. And so did the TI 99/4's product manager: when he saw what a mess the TI 99/4 had become, he quit TI and went to work for Atari, where he became the product manager for the Atari 400 & 800!

But even though computer experts realized that TI's computer was junk, TI decided to market it very aggressively, in several ways. TI coaxed Milton Bradley and Scott Foresman to write lots of programs for the 99/4. It paid researchers at MIT to make the 99/4 understand LOGO (a computer language used by young children and very popular in elementary schools). It improved the keyboard just enough so that people would stop laughing at it; the version with the new keyboard was named the *99/4A*. TI paid Bill Cosby to praise the 99/4A, and ran many hundreds of ads on TV that showed Bill Cosby saying "wow". TI dropped the price dramatically: the $1150 price was slashed to $650, then $150, and finally became just $99.50! (To bring the price that low, TI had to exclude the color monitor from the price; instead, TI included a hookup to your home's color TV.)

By contrast, Atari did hardly anything to market or further improve the Atari 400 and Atari 800. Instead, Atari concentrated on its other products: the big Atari game machines (which you find in video arcades), and the Atari VCS machine (which plays video games on your home TV).

The TI 99/4A therefore became more popular than the Atari 400 and Atari 800—even though the TI 99/4A was inherently worse.

SINCLAIR, OSBORNE, & BACKLASH

In 1980 and 1981, two important companies entered the microcomputer marketplace: Timex Sinclair (1980), and Osborne (1981).

The first complete computer that sold for less that $200 was invented by a British chap named Clive Sinclair and manufactured by Timex. The original version was called the *ZX-80* (because it was invented in 1980, contained a Z-80 CPU, and was claimed to be "Xellent"); it sold for $199.95. In 1981, Clive Sinclair invented an improved version, called the *ZX-81*. Later, he and Timex invented further improvements, called the *ZX Spectrum* and the *Timex Sinclair 1000*. When TI dropped the price of the TI 99/4A to $99.50, Timex retaliated by dropping the list price of the Timex Sinclair 1000 to $49.95, so that the Timex Sinclair 1000 remained the cheapest complete computer.

In April 1981, Adam Osborne began the Osborne Computer Corp. and began selling the Osborne 1 computer, which was designed by Lee Felsenstein, who had also designed Processor Technology's Sol 20 computer. The Osborne 1 computer cost $1795 and included practically everything a business executive needed: its $1795 price included a keyboard, a monitor, a Z-80A CPU, a 64K RAM, two disk drives, CP/M, Microsoft BASIC, a second version of BASIC, a fancy word processor (Wordstar), and an electronic-spreadsheet program for accounting (Supercalc). Moreover, the entire computer system (including even the monitor and disk drives) was collapsible, and turned itself into an easy-to-carry attaché case! It was the world's first portable business computer.

While Timex Sinclair and Osborne were entering the marketplace, Radio Shack, Apple, and Commodore were introducing new computers of their own. Let's look at them. . . .

In 1980, Radio Shack began selling three new computers. The *TRS-80 model 3* replaced Radio Shack's cheapest computer (the model 1) and was almost as good as Radio Shack's fanciest computer (the model 2). The *TRS-80 Color Computer* drew pictures in color and cost less than the model 3. The *TRS-80 Pocket Computer* fit into your pocket, looked like a pocket calculator, and was manufactured for Radio Shack by Sharp Electronics in Japan.

In 1980, Apple began selling the *Apple 3*. It was overpriced; and to make matters worse, the first Apple 3's that rolled off the assembly line were defective. Apple eventually lowered the price and also fixed the defects; but becuase the Apple 3 had gotten off to such a bad start, computer consultants didn't trust it and told everybody to avoid it.

In 1981, Commodore began selling the *VIC-20*, which drew pictures in color and cost less than Radio Shack's Color Computer. In fact, the VIC-20 was the first computer that drew pictures in color for less than $300. The VIC-20 originally sold for $299.95. When TI lowered the price of the TI 99/4A to $99.95, Commodore lowered the price of the VIC-20: at a typical discount department store (such as K Mart or Toys R Us or Child World), you could buy the VIC-20 for just $85: the VIC-20 was still the cheapest computer that could handle color. (The Timex Sinclair 1000 was cheaper but handled only black-and-white.) Moreover, the VIC-20 had standard Microsoft BASIC, whereas the Timex Sinclair 1000 and TI 99/4A did not; so the VIC-20 was the cheapest computer that had standard Microsoft BASIC; it was the cheapest computer that was pleasant to program. Also, the VIC-20 had a nice keyboard, whereas the keyboards on the Timex Sinclair 1000 and TI 99/4A were pathetic; so the VIC-20 was the cheapest computer that had a nice keyboard. The VIC-20 became immediately popular.

IBM PC

On August 12, 1981, IBM announced a new microcomputer, called the *IBM Personal Computer* (or *IBM PC*).

Although IBM had previously invented other microcomputers (the IBM 5100 and the IBM System 23 Datamaster), they'd been overpriced and nobody took them seriously—not even IBM. The IBM Personal Computer was IBM's first *serious* attempt to sell a microcomputer.

The IBM Personal Computer was a smashing success, because of its amazingly high quality and amazingly low price. It became the standard against which the rest of the microcomputer industry was judged.

In the years that followed, many other companies tried to imitate the IBM Personal Computer, but none of the imitations was competitive enough to hurt IBM.

RECENT DEVELOPMENTS

Today, the cheapest computer is still the Timex Sinclair 1000, which lists for $49.95. Timex also sells two souped-up models, called the *Timex Sinclair 1500* and the *Timex Sinclair 2068*. But Timex has announced it will quit selling computers soon.

Texas Instruments has announced it will stop selling the TI 99/4A. At the moment, TI is cleaning out its inventory, at very low prices. Instead of selling the TI 99/4A, TI will be selling the *TI Professional Computer*, which is much fancier, resembles the IBM Personal Computer, and costs about $2500.

Atari decided to replace the Atari 400 and Atari 800 computers by a new line of computers, called the *Atari 600 XL*, the *Atari 800 XL*, the *Atari 1400 XL*, and the *Atari 1450 XLD*. Atari has begun shipping the cheap ones (the 600 XL and 800 XL), but has postponed the more expensive ones—perhaps forever.

Commodore still sells the original VIC-20 for about $85, but also sells a souped-up version, called the *Commodore 64*, for about $200. Because the Commodore 64 has more RAM and better animated graphics and music than the VIC-20, most people buy the Commodore 64 instead of the VIC-20.

Radio Shack has improved its computers. The improved version of the model 2 is called the *model 12*; an even fancier improvement is called the *model 16*. The improved version of the model 3 is called the *model 4*; a version of the model 4 that's portable (and looks like an Osborne computer) is called the *model 4P*. The improved version of the Color Computer is called the *Color Computer 2*; Radio Shack has also invented a cheaper version, called the *Micro Color Computer*. The improved versions of the Pocket Computer are called the *Pocket Computer 2*, the *Pocket Computer 3*, and the *Pocket Computer 4*. Radio Shack also sells a new portable computer that fits in your lap; it's called the *Model 100*.

Apple replaced the Apple 2-plus by the *Apple 2e*. Apple still tries to sell the Apple 3, which is more expensive than the Apple 2e, and also sells the *Lisa*, which is *very* much more expensive. Apple is selling a new computer, called the *Macintosh*, which resembles the Lisa but costs much less. Two imitations of the Apple 2e are popular: one is the *Ace* (manufactured by Franklin); the other is the *Adam* (manufactured by Coleco). Apple sued Franklin, and forced Franklin to redesign the Ace to make it slightly less compatible. Coleco's Adam is different enough from the Apple 2e to avoid a lawsuit, but Coleco has had difficulty manufacturing the Adam reliably.

IBM has invented several souped-up versions of the IBM Personal Computer; the most important souped-up version is called the *IBM Personal Computer Extended* (or *IBM PC XT*). It's also invented a stripped-down version, called the *IBM PC Junior*. Many companies sell imitations of the IBM Personal Computer.

Hewlett-Packard has invented an exciting new computer, called the *HP 150*, which resembles the IBM Personal Computer but has a touch-sensitive screen: when your finger touches the screen, the computer knows where you're pointing and reacts appropriately.

Osborne Computer Company still exists but is bankrupt. Kaypro makes computers that resemble Osborne's but are better.

TODAY'S MARKETPLACE

Several new computers have come into the marketplace. Let's examine them.

Adam. The most controversial new computer is the Adam, by Coleco. Because many of the Adams shipped were unreliable, and because the additional hardware and software and manuals promised by Coleco aren't available yet, Coleco has developed a bad reputation. Independent software companies are reluctant to develop software for it. So although most of the Adams shipped today work okay, you'll have trouble getting software for them.

Because of Coleco's problems, cruel jokes about the Adam became popular. Here they are. . . .

1. To pronounce "Adam computer" correctly, put the accent on the *second* syllable: it's "A DAMN computer".

2. The Adam computer is made by Coleco, which also makes the Cabbage Patch doll. If your Adam doesn't work, put a wig on it and sell it as a "Cabbage Patch computer".

3. Each Adam has its own peculiar defect. In fact, Adam computers are like Cabbage Patch dolls: no two are alike.

Timex Sinclair 2068. To sell the new Timex Sinclair 2068 computer, Timex ran many full-page ads in computer magazines. The Timex Sinclair 2068 has nice color graphics, nice music, and a nice version of BASIC. The keyboard has a nice feel: it's manufactured for Timex by the Brother typewriter company.

But Timex's attempt to sell the Timex Sinclair 2068 failed, for several reasons. . . .

1. Timex's previous computer (the Timex Sinclair 1000) was crummy, and Timex had a hard time getting rid of the image of being a "crummy" company.

2. The Timex Sinclair 2068 is only slightly cheaper than the Commodore 64. The typical person would rather buy the Commodore 64, because more hardware and software has been invented for it.

3. The Timex Sinclair's non-standard keyboard layout confuses typists. For example, the quotation mark is on the P key! To type a quotation mark, you must press the P key while holding down the SYMBOL SHIFT key. The only other brand of computer that ever put the quotation mark on the P key was the Texas Instruments 99/4A—which is no longer being manufactured.

Radio Shack Color Computer 2. Many people have called Radio Shack's Color Computer 2 "the world's most undervalued computer". It has an excellent version of BASIC (much better than Commodore's and Apple's) and yet costs very little. I recommend it highly to people who want to write programs in BASIC. If you buy it, make sure to ask for Extended BASIC, not Standard BASIC.

Atari. To introduce young kids to computers, many schools teach the kids *LOGO*, which is a computer language that's easier and more fun than BASIC. LOGO is available on many microcomputers (such as the Texas Instruments 99/4A, the Commodore 64, the Radio Shack Color Computer 2, the Apple, and the IBM PC); but the nicest version for kids is Atari's. It works on any of Atari's computers and costs very little. An ideal computer for elementary schools might be the Atari 800 XL with Atari LOGO and Atari Microsoft BASIC 2.

IBM PC Junior. IBM's newest computer, the PC Junior, is a stripped-down version of the IBM PC. Its main disadvantage is its limited memory: you're limited to 128K of RAM and just one disk drive. Another disadvantage is its keyboard, which has fewer keys, unusual labeling, and an unusual feel. Some people *hate* the PC Junior's keyboard; others say "it's not so bad"; others think it's "fine". Try it yourself, and tell me what *you* think of it!

The original IBM PC was intended for business; the IBM PC Junior is intended for fun instead. Therefore, the IBM PC offers *more* colors and *fancier* music than the original IBM PC.

Sanyo MBC-550. If you want to buy an IBM PC, but can't afford it, and are therefore considering an IBM PC Junior instead, here's a more interesting alternative: buy the Sanyo MBC-550. For about $1000, you get everything you need: the computer itself, 128K of RAM, a Centronics printer interface, a disk drive, the MS-DOS operating system, and lots of software! And unlike the IBM PC Junior, the Sanyo MBC-550 is expandable: you can increase the RAM to 256K, and you can add a second disk drive.

Tandy 2000. The Tandy 2000 is Radio Shack's newest computer. It resembles the IBM PC but has higher-quality graphics, a higher-quality disk system, a higher-quality CPU, and a lower price! Check it out, and remember that you can pay less than list price by going to the Radio Shack discount dealers that I listed in volume 1.

DEC Rainbow. When shopping for a computer, remember that *the DEC Rainbow is the only low-cost computer whose screen displays 132 characters per line*. Although the list price of the DEC Rainbow is a little too high to be competitive, dealers are starting to offer generous discounts. Some dealers are including a printer at no extra charge. DEC is offering some schools a 70% discount, which makes the computer almost free! (DEC expects to make a profit from those schools, by selling them software and other add-ons at full list price.)

Hewlett-Packard 150. The Hewlett-Packard 150 uses MS-DOS (like the IBM PC, the Tandy 2000, and the Sanyo MBC-550). What makes the Hewlett-Packard's computer unusual is its screen, which is *touch-sensitive*: you communicate with the computer by pressing your finger against the computer's screen. Wordstar, Visicalc, and other popular programs have been modified, to make full use of Hewlett-Packard's amazing screen.

Unfortunately, the screen is slightly inaccurate: after you point to a place on the screen, you must tap the arrow keys on the keyboard, to fine-tune the position. Also, the computer is expensive: the hardware costs about $4000, and the software that lets you "point" costs about $500 per program. If the prices were slightly lower, the computer would be a big success.

Macintosh. Apple's newest computer is the Macintosh, which costs $2495. It's a stripped-down version of the Lisa. For the Macintosh, Apple has developed new, exciting software. Moreover, all the companies that invented good software for the IBM PC have announced that they're creating even fancier software for the Macintosh. By the fall of 1984, you'll be able to buy the Macintosh versions of BASIC, PASCAL, PFS, DBASE 2, 1-2-3, Multiplan, and all the other popular software packages—and the Macintosh versions will be fancier than the versions that are on the IBM PC, so that the Macintosh will become the world's most useful computer.

Phone for an update. I wrote those comments at the beginning of 1984; but every day, the computer industry changes. To find out *today's* news about any particular company, phone me at 617-266-8128, and I'll tell you the news that's reached my ears. Since people who read this book feed me inside info about many companies, I hear practically everything that's going on.

MYTHS

CP/M

Myth: microcomputers that use the CP/M operating system can all use the same software.

Reality: even if your microcomputer uses the CP/M operating system, it might not be able to run any of the popular CP/M software! For example, the CP/M that Commodore sells for its Commodore 64 computer handles only *two* software packages! The CP/M that DEC provided for DEC's VT-180 "Robin" computer handled only four software packages. Here's why. . . .

Suppose I write a program for my CP/M computer, put the program onto a CP/M disk, and then hand you the disk. Even if your computer can "handle CP/M", it might not be able to handle the disk I gave you, for the following reasons. . . .

The disk I gave you might be the wrong size. Maybe I gave you an 8-inch disk, whereas your computer requires a 5¼-inch disk instead.

The disk I gave you might have the wrong format. Maybe I gave you a double-sided disk, whereas your computer accepts only single-sided. Maybe I gave you a double-density disk, whereas your computer accepts only single-density. Maybe I gave you a disk that has wide sectors (512K per sector), whereas your computer accepts only disks that have narrow sectors (256K per sector or even 128K per sector). Maybe I gave you a disk whose directory is on track 2, but your computer expects the directory to be on a different track instead.

The program I wrote might assume the computer's screen can show 25 lines of 80 characters, whereas your computer's screen shows only 24 lines instead of 25, or shows only 40 characters per line instead of 80.

The program I wrote might include graphics commands that *your* computer can't handle. The program I wrote might be written for a certain brand of printer that you don't own.

If you buy a computer that uses CP/M, ask which other computers it's compatible with; and try to make the salesman prove it (by removing a disk from one computer and sticking the same disk into the computer he's trying to sell).

For example, some computers are advertised as using "CP/M with an Osborne-compatible format"; such computers can run *most* (though not all) of the CP/M disks that have been developed for Osborne computers.

Also, remember that there are two main types of CP/M. The old type (called CP/M-80) requires that the computer's CPU be an 8080 or 8085 or Z-80. The new type (called CP/M-86) requires that the CPU be an 8086 or 8088. The old type is *not* compatible with the new type.

There are also several other variations, such as *Concurrent CP/M* and *MP/M*, which are similar but—alas!—incompatible.

So the idea that "CP/M is a standard" is a myth. If you buy a CP/M disk and stick it into your CP/M computer, the disk will probably *not* work. It will work only if the disk happens to use the same format, version, etc. as your computer requires.

MS-DOS

Myth: if you buy a computer that uses the MS-DOS operating system, it can run all the software that was written for the IBM PC.

Reality: MS-DOS is almost as loose a "standard" as CP/M.

POWER

Myth: to tell how "powerful" your computer is, look at how much RAM it has. Similar myth: to tell how "powerful" your computer is, look at which kind of CPU chip it contains; if it contains a "16-bit" CPU, your computer is more powerful than if it contained an "8-bit" CPU.

Reality: RAM chips and CPU chips are cheap and relatively unimportant. Today, RAM chips cost only 70¢ per K, and CPU chips cost between $3 and $20. The most expensive—and most important—parts of computer system are the disk drives, printer, screen, keyboard, interfaces, ROM chips, and—most important of all—the software. To judge an entire computer system by looking at just its RAM and CPU is as stupid as judging an entire car by looking at just the number of cylinders in its engine; if lots of cylinders were all that mattered, V-12 cars would be very popular.

A sucker is the guy who buys 512K of RAM for his IBM PC so that he can write long programs in BASIC—and then discovers that IBM's BASIC is unable to access more than 64K! A sucker is the guy who buys a Coleco Adam because it has 80K (which sounds like a lot)—and then discovers that the Adam's CPU can handle only 32K of RAM when doing word processing, and—since Adam's BASIC is poorly written and sits in RAM instead of ROM—leaves only 26K for his BASIC programs!

A fool is the guy who pays the extra bucks for a new super-fast CPU—and then buys a slow printer, a slow disk drive, can't use the computer because he's always waiting for the repairman, can't write programs quickly because he can't decipher the manual, and spends most of his life yelling at the manufacturer to deliver the software that the manufacturer promised but didn't finish yet.

APPLE

SLOTS

If you open an Apple 2 or 2-plus or 2e, you'll see *slots*. Into each slot, you can put a printed-circuit card.

When Steve Wozniak invented the Apple, he intended the slots to be interchangeable: you could put any card into any slot. But over the years, traditions have developed, about which card to put into which slot. Today, most software assumes you followed the traditions, which are as follows. . . .

The original Apple 2 and 2-plus each contained eight slots, numbered from 0 to 7. They also contained 48K of RAM. In slot 0, put a card containing 16K of extra RAM (to bring the total RAM to 64K). In slot 1, put a card containing a Centronics parallel printer interface. In slot 2, put a card containing an RS-232 serial interface (to attach to your modem and telephone). Normally, the Apple's screen shows only 40 characters per line; if you want 80 characters per line instead, you must put an *80-column card* into slot 3. In slot 6, put a card containing a disk-drive controller; it controls two disk drives. In the remaining slots (4, 5, and 7), you can insert any cards you wish, such as a voice synthesizer, a music synthesizer, a CP/M card, a clock, or a controller for an additional pair of disk drives.

The Apple 2e is slightly better. Because it already comes with 64K of RAM on the motherboard, slot 0 isn't needed, and has therefore been omitted. The Apple 2e includes *two* versions of slot 3: one version is the traditional version (for use with traditional cards); the other version (which I'll call "slot 3A") can hold a dual-function card. The dual-function card enables 80 columns and also contains 64K of extra RAM (bringing the total RAM to 128K). For slot 3A, Microsoft sells an even fancier card—a *three*-function card—which enables 80 columns, contains 64K of extra RAM, and also contains a Z-80 CPU, so that you can run CP/M. Microsoft's three-function card lists for $495; its price includes a disk that contains CP/M and Microsoft BASIC version 5; its called the *Microsoft Premium Softcard 2e*.

Whenever you buy an Apple, make sure you understand which cards the salesman is omitting.

APPLE'S REPUTATION

Apple Computer Company was founded by two friendly, cheery, nice, young engineers: Steve Wozniak and Steve Jobs. Wozniak invented the machine; Jobs worried about marketing and production. Afterwards, Wozniak got into an airplane crash which hurt his head and gave him amnesia, so he left the company and enrolled in college under a fake name ("Rocky Clark"). After he graduated, he returned to Apple Computer Company, where he helped develop Apple's new Macintosh computer.

Though Wozniak and Jobs are wonderful guys, their marketing strategies have occasionally raised a few eyebrows. For example, Apple's ads have claimed that the Apple was the first personal computer (it was *not* the first!); Apple launched a big campaign to make businessmen buy Apple PASCAL (even though Apple PASCAL is of no help to the average businessman whatsoever); Apple prohibited its dealers from displaying games (though Apple has since relented); and Apple still prohibits dealers from selling Apples via mail order.

Apple Computer Inc. (as it's now called) has been donating free Apple computers to schools, for three reasons: to be nice, to get a tax write-off, and to lure schools into buying more Apples (to be compatible with the Apples that the schools received free). But if Apple *really* wanted to be nice, it would lower the Apple's price, or sell a stripped-down system that low-income consumers could afford. At the moment, Apple isn't trying to sell to the poor; instead, Apple is trying to sell to the "chic". Apple's Lisa and Macintosh are both examples of chic computers that only the rich can afford.

Although the Lisa and Macintosh are somewhat expensive, they're wonderful. They're setting new standards of quality, which the rest of the computer industry will try to imitate.

COMMODORE

WHY COMMODORE?

Commodore manufactures the *VIC-20*, which is the world's cheapest decent computer. (The TS 1000 is cheaper, but contains too little ROM and is therefore considered *in*decent.)

You can buy the VIC-20 at a discount department store (such as K-Mart or Toys-R-Us or Child World) for just $85. But even though it's sold by toy stores, it's a real computer, and it can really do useful work!

Besides the VIC-20, Commodore makes many fancier computers, such as the *Commodore 64*, the *PET*, and the *Commodore Business Machine (CBM)*.

In my opinion, the four most important computer companies are Commodore, Radio Shack, Apple, and IBM. Whenever anybody asks my advice about which computer to buy, I say to begin by considering those four companies. Of those four companies, Commodore charges the least.

HOW COMMODORE BEGAN

Before Commodore made computers, Commodore made electronic pocket calculators. Each calculator contained a microprocessor (a one-chip CPU). The microprocessor was manufactured by a company called MOS Technology.

But MOS Technology was in trouble, because of how that company had begun. In 1974, the most popular microprocessors had been the "8080" (manufactured by Intel) and the "6800" (manufactured by Motorola). In 1975, Chuck Peddle, who helped invent the 6800, quit his job at Motorola and started a new company with his friends. That new company was MOS Technology. It began manufacturing a microprocessor called the "6501", which was so similar to Motorola's 6800 that Motorola threatened to sue. To try to prevent the lawsuit, MOS Technology stopped manufacturing the 6501, and instead began manufacturing the "6502", which Chuck Peddle invented, and which acted differently enough from Motorola's 6800 to avoid offending Motorola. The 6502 chip became very popular; many companies paid MOS Technology to manufacture the 6502 chips. One of MOS Technology's biggest customers was Commodore, whose headquarters were in Canada then.

Even though the 6502 chip was legal, Motorola decided to sue MOS Technology for its illegal predecessor, the 6501. In the courts, the case dragged on for two years and cost MOS Technology many thousands of dollars in lawyers' fees. Finally, in 1977, Motorola won, to the tune of $200,000.

That put MOS Technology into financial trouble, so it announced it wanted to be bought by some rich company that had lots of cash. Commodore (a rich company) bought it.

Shortly before that sale, Canada's tax laws changed—to Commodore's disadvantage. So Commodore moved its headquarters (in theory) to the Bahamas. Commodore's profits were then taxed by the Bahamas, where the tax rate is very low.

So MOS Technology became part of "Commodore Limited", a Bahamas company. And that's how Commodore had suddenly found itself running a company that manufactures chips. Commodore had entered the computer business.

DEALING WITH COMPETITORS

At MOS Technology, Chuck Peddle had sold a 6502 chip to Steve Wozniak for $25. Steve used the chip to create the Apple computer. Steve Wozniak and his friend Steve Jobs started selling the Apple computer, and called themselves the Apple Computer Company. The Apple computer became quite popular in California. (Many years later, it would become the most popular microcomputer company in the *world*!)

Commodore saw that the Apple was cute and profitable in California, and offered to buy the company. Commodore almost succeeded. But when Commodore and Apple haggled over the price, they couldn't quite agree: Apple wanted $15,000 more than Commodore was willing to cough up. So Commodore didn't buy Apple (and has regretted it ever since).

Then Commodore hired Chuck Peddle to design a "Commodore computer". Commodore hoped to sell the Commodore computer to Radio Shack, and let Radio Shack distribute the computer through Radio Shack's stores. Radio Shack said, "Yes, we like your idea. Finish designing your computer, and tell us more about it." Commodore finished designing its computer and showed it to Radio Shack. Radio Shack said "Thanks, we liked your idea of a low-cost computer so much that we've designed our own. Thanks for the idea." That's how Radio Shack got the idea of developing the TRS-80 computer!

HOW COMMODORE LAUNCHED ITS PET

Having unsuccessfully tried to buy Apple, and having unsuccessfully tried to make Radio Shack sell Commodore's computer, Commodore realized that the only way to sell computers was to go directly to the American public. So in 1977, Commodore publicly announced it was going to sell its own computer.

Commodore called its new computer the *PET*, because Commodore's head of marketing was the same guy who invented the Pet Rock. He figured, "If people are stupid enough to love a Pet Rock, they'll love a Pet Computer even more!" He was right: people fell in love with the idea of a Pet Computer and mailed lots of money to Commodore.

Commodore told the press that "PET" was an abbreviation for "Personal Electronic Transactor". But actually, Commodore invented the name "PET" first, and later concocted what "PET" could be an abbreviation for.

Originally, Commodore said the PET would cost just $495. That price would include *everything*: the CPU, the RAM, the ROM, the keyboard, the monitor, and even the tape recorder. The ROM would have a good version of BASIC. The screen would display upper-case letters, lower-case letters, punctuation, mathematical symbols, and many weirder symbols also (such as hearts, diamonds, clubs, spades, curves, circles, and rectangles).

That frightened the other manufacturers of personal computers. Commodore's price was far lower than everybody else's. Commodore's computer offered far more features. And Commodore's wealth insured that Commodore could spend far more on advertising, marketing, and promotion than all the other manufacturers combined.

Many personal-computing magazines praised the PET and heralded it as "the birth of a new generation" in personal computers. Chuck Peddle, the guy who designed the PET, was treated to many interviews.

Commodore raised its price from $495 to $595 before taking orders. To order the PET, the customer had to send the $595 in full, plus shipping charges, and then wait for Commodore to deliver.

Many people mailed Commodore the money and waited . . . and waited . . . and waited. Commodore didn't ship. People got impatient.

Computer stores got annoyed. The stores had advertised that they'd be selling Commodore's PET, and had received many orders from customers, but Commodore wasn't yet delivering to the stores. Customers complained to the stores, and the stores couldn't do anything to solve the problem.

While all that was going on, Radio Shack entered the market with its TRS-80 model 1, which was priced originally at $599—about the same price as Commodore's PET. Moreover, Radio Shack had better policies. To order, you had to give Radio Shack just a 10% deposit (Commodore required prepayment in *full*); Radio Shack didn't charge for shipping (Commodore did); Radio Shack set up repair centers throughout the country (Commodore's only repair center was in California); Radio Shack delivered computers to customers quickly (Commodore still wasn't delivering!).

Finally, Commodore made an announcement about delivery: Commodore announced that the $595 PET would *not* be delivered in the near future! Instead, Commodore would deliver a $795 version, which would include 4K of extra RAM. So if you already sent $595 to Commodore and wanted a computer soon, you'd have to send an extra $200.

That was a rip-off. 4K of extra RAM is *not* worth an extra $200; it's worth much less. But customers were so desperate that they sent the $200 anyway.

Radio Shack shipped its computers on a first-come first-served basis. Commodore didn't: Commodore gave preferential treatment to its "friends". If you ordered a computer from Radio Shack, you could predict when it would arrive: Radio Shack would give you an accurate estimate. If you ordered a computer from Commodore, you hadn't the faintest idea of when it would arrive, because you didn't know how many "friends" were on Commodore's list.

When you ordered a computer from Radio Shack, you received a 232-page well-written easy-to-read cheery manual, written by David Lien and attractively bound. When you ordered a computer from Commodore, you received just 10 loose pages, which were tough to understand and were incomplete.

Commodore announced a low-cost printer; then Commodore changed its mind, and decided to sell only a more expensive printer instead. Commodore announced a low-cost disk drive; then Commodore changed its mind, and decided to sell only a more expensive unit instead, which contains *two* disk drives. Those misleading announcements made the public's confidence in Commodore dip even further.

That's how Commodore entered the personal computer market. And that's why the PET (which people thought would be the best-selling computer) managed to drop in popularity, below Radio Shack, and below even Apple.

As Commodore's fortunes dipped, Chuck Peddle and his friends quit, and were hired by Apple. But at Apple, they were treated as second-class citizens, so they returned to Commodore.

THE PROBLEM WITH RAM

Commodore came out with several versions of the PET. Each version contained a different quantity of RAM.

If you bought a version that contained little RAM and later wanted to increase the RAM, Commodore refused to install extra RAM. Instead, Commodore insisted that you buy a whole new PET.

Some of Commodore's customers, who couldn't afford a new PET, bought extra RAM chips from electronics supply houses and installed the RAM chips themselves.

Commodore disliked that tinkering, and decided to stop those tinkerers. So Commodore cut a hole in the PC board, exactly where the extra RAM chips would go, so that you couldn't insert the extra RAM chips.

That nasty hole angered Commodore's customers, especially since it cost Commodore almost as much money to cut the hole as it would have cost to insert the extra RAM chips!

THE PROBLEM WITH TAPE

Commodore changed the PET slightly, so that it handled tapes differently. The new PET tape system was incompatible with the old PET tape system: tapes created for the old PET wouldn't work on the new PET. Commodore neglected to tell its customers about the change. Moreover, the new PET looked exactly like the old PET: it didn't contain any kind of label saying "new". So all of Commodore's customers got confused. Customers who wrote programs for their old PETs and then bought additional PETs discovered that their programs wouldn't work on the new PETs. They thought their new PETs were broken.

Many companies had been selling computer programs for the PET on tape. When Commodore changed the tape system without telling those companies, the companies started receiving angry letters from customers who had bought the tapes, found the tapes didn't work on their new PETs, and accused the companies of selling junk tapes that didn't work. The customers thought the companies were rip-off artists; the companies thought their customers were lying; and it took a long time for everybody to realize that the real culprit was Commodore, who had changed the PET without telling anybody.

When the companies discovered that Commodore had changed the PET and also discovered that there was no label to distinguish the new PETs from the old PETs, the companies realized they'd have to give *two* copies of each program to each customer, so that the customer could try both versions to determine which PET the customer had. At that point, many companies gave up trying to sell PET tapes. They sold tapes for Apple and Radio Shack computers instead. That's why there's more software for Apple and Radio Shack than for Commodore.

SUCCESS IN SCHOOLS

In spite of all those hassles, Commodore's PET computer sold well, because the PET's price was low and its quality was high. Commodore gave a special deal to schools: if a school bought two PETs from any computer store, Commodore would ship the school a third PET free! That "three for the price of two" policy encouraged many schools to buy the PET, because it made the price per PET far less than for any other company's computer. The PET became the most popular computer in schools. (It became more popular than even Radio Shack and Apple.) Many kids fell in love with "the teacher's PET".

Years later, Radio Shack and Apple both outsold the PET (even in schools), because of the PET's lack of tape software, and because Radio Shack and Apple disk drives are easier to use than Commodore's, and because Radio Shack and Apple improved their own computers significantly whereas Commodore did not, and because Radio Shack and Apple also started offering discounts to schools.

SUCCESS IN EUROPE

The PET became the best-selling computer in England and in several other European countries. That's because Commodore's marketing people in Europe were nice. (Commodore's marketing people in the United States were jerks.)

In England, Commodore encouraged customers to write programs for various industries. For example, customers wrote programs that made the PET handle insurance. Commodore publicized those programs and helped its customers sell them to other customers. A whole network of friendly Commodore users arose. Too bad that happened only in England and Europe!

Commodore's share of the computer market was slowly sinking. But in 1980, an event occurred which totally changed Commodore's fortunes.

In April 1980, Commodore's founder (Jack Tramiel) told his top executives that he wanted Commodore to produce a computer for under $300—because if Commodore didn't, the Japanese would. Moreover, he wanted the computer to use color. At that time, the only computer for less than $300 was Sinclair's ZX-80, but it was black-and-white and crummy.

MOS Technology (which was owned by Commodore) had already invented a Video Interface Chip (VIC); that single chip could handle the entire process of sending information from the computer to the TV screen. Because that chip had already been invented and was so cheap, Commodore decided to use it in the under-$300 computer.

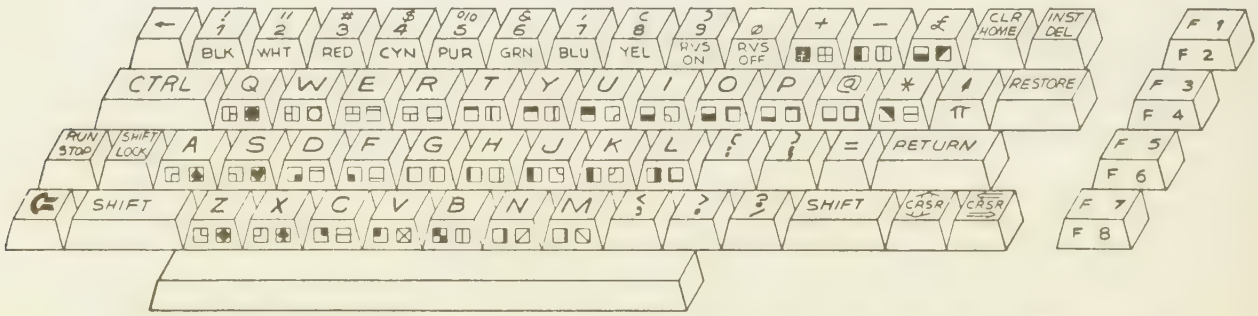
Unfortunately, the chip put only 22 characters per line on the screen. (By contrast, the PET had 40 characters per line, and most computers today have 80 characters per line.) So the under-$300 computer would have only 22 characters per line.

Commodore wanted to call the new computer the "Vixen" (which is a female fox). But a "Vixen" computer wouldn't sell well in Germany, because it sounds like a certain German obscene word. So Commodore decided to call the computer the "VIC", to honor of the Video Interface Chip inside it. (Commodore didn't realize that in German, "VIC" sounds even more obscene than "Vixen"!)

But Commodore felt that the name "VIC", though cute, was too short. Since the computer would print 22 characters per line, Commodore changed the name to "VIC-22". But Commodore wanted the computer to seem friendly, and the number 22 was too "unfriendly" and hard to remember; so Commodore changed the name to "VIC-20". Today, the computer's called the "VIC-20", except in Germany (where it's called the "Volks Computer VC-20") and in Japan (where it's called the "VIC-1001").

Commodore began shipping the VIC-20 in 1981. The original price was $299.95; but over the years the price has dropped, so that today you can buy it for just $85 at discount department stores such as K-Mart, Toys-R-Us, and Child World.

Unlike Timex Sinclair computers, the VIC-20 has real moving keys. The keyboard is fancy:



The keyboard is easy to understand. For example, look at the S key. On the front of that key, you'll see two pictures: the right picture is a heart; the left picture is part of a box. If you press the S key, you normally get a capital S. To get the heart instead, hold down the SHIFT key. To get the partial box, hold down the Commodore key (which is left of the left SHIFT key).

That's how the keyboard normally works. But you hold down the SHIFT key and the Commodore key simultaneously, you make the keyboard switch to *typewriter mode*. Once you're in typewriter mode, pressing the S key gives you a *lower*-case S (like an ordinary typewriter); to get a capital S, you hold down the SHIFT key; you can still get the partial box by holding down the Commodore key.

Normally, the computer prints the letters on the screen in purple (on a white background). If you don't like purple, you can choose a different color. The top row of keys contain the colors: you can choose black, white, red, cyan, purple, green, blue, or yellow. For example, if you want to switch to red (instead of purple), tap the RED key while holding down the CTRL key.

The computer prints dark letters on a white background, usually. You can make the computer switch to white letters on a dark background, by tapping the RVS ON key while holding down the CTRL key.

The computer can make the TV play music: three-part harmony, plus percussion!

The computer includes a 6502 CPU, a 16K ROM (which handles Microsoft BASIC), and a 5K RAM. (Of the 5K RAM, 1½K is used for system overhead, which leaves 3½K for your own program.) You can buy a 16K RAM cartridge, which expands the RAM to a total of 21K.

You can't use your home's tape recorder; you must use Commodore's, which lists for $75. For a disk drive, discount dealers now charge just $250.

I recommend the VIC-20, because it's a decent computer at a low price (just $85). My only complaints are the following. . . .

Commodore charges too much for the tape recorder and too much for the 16K RAM cartridge. The image on the TV screen is fuzzier than the image displayed by other manufacturers. If you make a mistake when you're typing your program, you'll be tempted to use Commodore's built-in editor, but the editor is peculiar and often does you more harm than good. The computer's ROM contains one error, which prevents the computer from successfully handling INPUT statements that contain long questions. If you can't easily draw a shape that's not on the keyboard, because the computer doesn't understand coordinates—unless you buy Commodore's *Super Expander Cartridge*, which contains more RAM and ROM and costs $70. The commands that handle the disks are awkward. Commodore's quality control is poor: many of the VIC-20's being shipped come with problems such as a noisy power supply (you hear a buzz), a broken TV-channel switch, etc. Of all the popular computers on the market today, the VIC-20 survives rough handling the poorest: during transportation, the keys break off, and you cannot easily repair them. If your computer needs repair after the 90-day warranty has expired, you must mail the computer to Commodore with $55.

Because of those problems, the VIC-20 is not a superior machine. But it's the best machine *in its price range*.

To sell the VIC-20, Commodore has tried three different kinds of ads. The first kind featured TV star William Shatner (who played Captain Kirk in Star Trek); the ad emphasized how the VIC-20 was wonderful, amazing, out of this world, fun, exciting. But then people started thinking of the VIC-20 as just a sci-fi toy. To combat the "toy" image, Commodore changed to its second kind of ad, which said that the VIC-20 was the same price as a video-game machine and yet was much more educational for your kids. When Texas Instruments began making similar claims, Commodore changed to its third kind of ad, which emphasized that Commodore's disk drives, printers, and telephone hookups cost much less than Texas Instruments'.

By using those three kinds of ads, Commodore has sold *more than a million* VIC-20's.

COMMODORE 64

In 1982, Commodore began selling an improvement of the VIC-20. The improved version is called the *Commodore 64*, because it includes 64K of RAM. (The original VIC-20 had only 5K.) The Commodore 64 also includes 20K of ROM. (The original VIC-20 had only 16K.) The Commodore 64 displays 40 characters per line. (The original VIC-20 displayed only 22 character per line.)

The Commodore 64 costs more than the VIC-20. The price of the Commodore 64 has gone through three phases. . . .

In the first phase, the recommended list price was $595, and Commodore tried to force all its dealers to charge that price in full. If a dealer advertised a discount, Commodore stopped shipping computers to that dealer. (That practice, which is called *price fixing*, is illegal. But in the computer field, nobody cares about laws.)

In the second phase, Commodore permitted discounts—many dealers charged just $350—and Commodore also mailed a $100 cash rebate to anybody who traded in another computer or a video-game machine, even if that computer or game was worth less than $100. For example, many people bought the TS 1000, just for the purpose of trading it in to buy a Commodore 64. In New York City, a Commodore dealer called "Crazy Eddy" sold junky video-game machines for $10, just so that his customers could mail them to Commodore to get the $100 rebate. That unusual rebate policy earned Commodore lots of attention—which is what Commodore wanted! Commodore donated all the traded-in computers and games to charities and got a huge tax write-off.

Now Commodore is in its third phase. The $100 cash rebate has been discontinued, but has been replaced by a lower over-the-counter price. Now you can get the Commodore 64 at Child World (and similar stores) for just $200! And the disk drive, printer, and color monitor cost just $250 each! Moreover, the drive, printer, and color monitor also work with the VIC-20.

Because of those new low prices, the Commodore 64 is becoming extremely popular. But be careful. Although the price is low, and although the machine is decent, some of Commodore's advertising is misleading. Let's see how. . . .

Commodore's ad says that the Commodore 64 has "64K of RAM". But when you turn on the computer, the first message you see on your screen says that you have only 38K left! The 26K that "disappeared" was used up by "system overhead".

Commodore's ad says that the Commodore 64 can handle "3D graphics". But actually, the computer doesn't even know how to draw a cube! The only thing "3D" about Commodore's graphics is that if you try to put two objects at the same place on the screen at the same time, the computer will hide one of the objects behind the other.

The Commodore 64 can produce some pretty graphics. It can also produce high-quality music: it lets you fine-tune each note's pitch and tone quality. But although you can produce nice graphics and music, doing so is extremely complicated. Moreover, Commodore's manual is too complicated for a beginner to understand. If you want to buy a computer that lets you produce high-quality graphics and music *easily*, you should buy an easier, more pleasant computer instead: buy an Atari or a Radio Shack Color Computer or—if you can afford it—an IBM Personal Computer.

Commodore claimed that the Commodore 64 would handle CP/M. What Commodore "neglected" to mention was that even though the Commodore 64 would handle CP/M, it wouldn't run any of the popular CP/M programs! That's because all the popular CP/M programs require 80 characters per line, and the Commodore 64 produces only 40 characters per line. Commodore plans to create its *own* CP/M programs that produce 40 characters per line; but Commodore's programs will be for games and education, *not* for business. Moreover, customers have been waiting about a year for Commodore's CP/M, and Commodore still hasn't delivered it!

The original version of the Commodore 64 created fuzzy characters on the screen. In February 1983, Commodore improved the Commodore 64; the improved version creates sharper characters. To create the sharper characters, Commodore had to change the ROM. Unfortunately, some programs written for the old ROM don't work with the new ROM. For example, Commodore's been selling software that converts programs from the PET to the Commodore 64; but that software works only if you have an *old* Commodore 64. Commodore's been distributing educational programs for schools; but 20% of those programs work only on the *old* Commodore 64. Commodore didn't tell its dealers or customers about those problems. As a result, many customers are buying Commodore's software, discover it doesn't work on the new Commodore 64, and feel ripped-off.

In Commodore's ads, Commodore claims that the Commodore 64 is more "powerful" than higher-priced computers (such as the Apple and the IBM Personal Computer), because the Commodore 64 has 64K whereas the other computers have fewer K. That claim is extremely misleading, for four reasons. First, the current versions of the Apple and the IBM Personal Computer *do* have 64K; the information in Commodore's ads is outdated. Second, RAM is cheap: you can buy 64K of RAM for just $64; and you can add more RAM to an Apple or an IBM Personal Computer whenever you please. Third, although the Commodore 64 has 64K of RAM, only 38K is available for you to use; the Apple and the IBM Personal Computer offer much more available K. Third, and most important, a computer's RAM is *not* a measure of the computer's power: you should also consider the ROM, the keyboard, the screen display, the disk system, the variety of available software, etc. In fact, compared to an IBM Personal Computer, a Commodore 64 is junk.

On the other hand, a Commodore 64 costs very little. When you compare it against other computers in its price range (such as the Atari 800 and the Radio Shack Color Computer), the Commodore 64 is "okay", though not superior. The Commodore 64 is a fine computer, but Commodore should stop producing misleading ads. How can Commodore employees sleep well at night, when they realize they work for such a dishonest company? I hope Commodore continues to produce good computers but more honestly.

COMMODORE TODAY

Today, Commodore's biggest seller is the Commodore 64. Its second biggest seller is the VIC-20.

Commodore still sells the PET, though the version it sells today is slightly different from the original version. Commodore also sells the *Commodore Business Machine (CBM)*, which resembles the PET but is fancier.

Commodore frequently announces new machines and products. Whenever you hear a Commodore announcement, don't believe it. Wait and see. Usually, the product that Commodore announces *does* eventually get manufactured, but the delay between the announcement and the manufacturing is usually about a year. Also, Commodore products tend to be nice, but never *quite* as nice as the ads imply.

Commodore has announced dozens of new products for 1984, but it's too early to tell which of Commodore's products will succeed, which will flop, and which will never get built.

RADIO SHACK

HOW THE TRS-80 BEGAN

The Tandy Leather Company was begun by Charles Tandy. Later, he acquired Radio Shack, which had been a Boston-based chain of discount electronics stores.

Under his leadership from his Fort Worth headquarters, Tandy/Radio Shack succeeded and grew 30% per year, helped by the CB radio craze that was sweeping the country. When the market for CB radios declined, he began looking for a new product to sell, to continue his 30% growth.

Commodore was inventing a computer and tried to convince Tandy's staff to sell it. Don French, a Tandy salesman whose hobby was building computers, recommended to Charles Tandy that Radio Shack start selling computers. Instead of buying computers from Commodore, Radio Shack hired Steve Leininger to design a Radio Shack computer and to keep the cost as low as possible.

Steve wanted his computer to be able to handle lower-case letters, but because interfacing the lower-case chip would have added 10¢ to the cost, management rejected lower-case: so Radio Shack's computer handled only capitals. (In those days, lower-case letters weren't considered important. Later, when customers began demanding lower-case letters, Radio Shack regretted not having spent the extra dime. Customers were spending $50 to rip open the Radio Shack and rearrange the chips to get lower-case.)

What color should the computer be painted? Radio Shack chose "Mercedes silver"—so that when you get your hands on the computer, you'll think you're driving a Mercedes.

(Unfortunately, since Mercedes silver looked like gray, Radio Shack became nicknamed "the great gray monster", and so artists switched to Apples, whose beige better matched their living-room decors. Later, in 1982, Radio Shack wised up, and switched from "Mercedes silver" to white.)

Radio Shack's computer listed for just $599 and consisted of four boxes:

a keyboard (inside of which hid the CPU, ROM, and RAM)
a monitor (manufactured for Radio Shack by RCA)
a cheap Radio Shack tape recorder
an AC/DC transformer

Wires ran from those boxes to each other, so that the whole system looked like an octopus.

Radio Shack wanted to put the AC/DC transformer *inside* the keyboard, so that the computer system would consist of three boxes instead of four. But if the AC/DC transformer had been inside the keyboard, Underwriters Laboratories would have delayed approval for 6 months, and Radio Shack didn't want to wait that long.

Radio Shack named its computer the *TRS-80*, because it was by Tandy's Radio Shack and contained a Z-80 CPU.

To official announce its new computer, Radio Shack called a meeting of the press. The meeting was to take place on a Monday morning in August 1977, on the front steps of the New York Stock Exchange. On Monday morning, sure enough, Radio Shack's management was standing on the steps, surrounded by reporters. Then suddenly, a reporter ran up to the group and yelled out that a bomb had gone off, two blocks away. The reporters all ran away to cover the story. And Radio Shack wasn't able to announce its new computer!

Radio Shack needed to quickly find a new place to announce its computer. Radio Shack heard that the Boston Computer Society was going to run a computer show that week—Wednesday through Friday. So Radio Shack's management drove up to Boston, got a booth at the show, and announced its computer there. Imagine Radio Shack's surprise, when the management discovered that the entire show and the entire Boston Computer Society was run by Jonathan Rotenberg, who was just 14 years old!

The introduction was successful: people liked and bought Radio Shack's new computer.

(In future years, many companies followed Radio Shack's footsteps: they announced new computers at meetings of Jonathan's Boston Computer Society.)

THE MILLERS

One of Radio Shack's first customers was Dick Miller. On the second day of the Boston Computer Society show (Thursday), he bought a TRS-80 from Radio Shack's booth, because a computer system with two disk drives and a printer would cost a total of only $2600 from Radio Shack, whereas the cheapest price that Dick had been able to find elsewhere was $4600.

Dick was a physicist and environmentalist. His wife Jill was a professional COBOL programmer who dabbled in a dozen languages. Together, Dick & Jill started a business, in which they rented Radio Shack computers to the general public.

They charged $30 for a weekend, $60 for a five-day week. Their motto: "Try before you buy!"

Radio Shack stores sent potential customers to Dick & Jill, to whet the customers' appetite by renting them computers. Since Dick & Jill knew more about programming than Radio Shack's own salesmen, Dick & Jill soon became highly paid consultants.

Radio Shack hired Dick & Jill to train Radio Shack's own staff, and gave Dick & Jill the newest equipment as soon as it was available.

Radio Shack hired Randy Cook to write the DOS. Dick was one of the first people to try it. He noticed that DOS version 1.0 didn't work; it didn't even boot! He notified Radio Shack, which notified Randy Cook, who then fixed that problem and wrote version 1.1. Dick noticed that it worked better but still had one major flaw: it didn't tell you how much disk space was left and--even worse--as soon as the disk was entirely filled it would self-destruct!

Then came version 1.2, which worked better, though still not perfectly.

(Because Radio Shack's DOS was still buggy, the inventors of Visicalc decided to put Visicalc onto the Apple instead of onto the TRS-80. Apple became known as "the Visicalc machine", and many accountants began buying Apples.)

Meanwhile, an independent Colorado company named *Apparat* was inventing its own improvements to Radio Shack's DOS. Apparat showed its improvements to Dick, who liked them and recommended that they be called "NEWDOS". That's how NEWDOS was born. Many of Radio Shack's customers bought NEWDOS and formed NEWDOS colonies.

Years later, Dick & Jill grew frustrated at Radio Shack's BASIC, because it ran too slowly and didn't permit structured programming. Several companies promised to develop a faster version of BASIC, a wonderful version of PASCAL, and a wonderful version of FORTH, but none of those promises came true. Finally, Dick & Jill stopped waiting: they hired a programmer to write a wonderful version of FORTH for them. Dick & Jill were so happy with FORTH that they stopped renting computers, stopped writing programs in BASIC, and now devote all their energies to working with their version of FORTH, which is called *Miller Microcomputer System FORTH*. That version of FORTH, which is excellent, runs on Radio Shack computers and also on the IBM PC. To buy it, contact Dick & Jill Miller at 617-653-6136. (Warning: although Dick thinks FORTH is easy, it's harder than BASIC, so I don't recommend it for beginners.)

DEALING WITH THE PUBLIC

In 1977, when Radio Shack began selling the TRS-80, customers didn't understand what computers were.

For example, at a Radio Shack show, I saw a police chief buy a TRS-80. As he carried it out of the room, he called back over his shoulder, "By the way, how do you program it?" He expected a one-sentence answer.

Radio Shack provided a toll-free 800 number for customers to call, in case they had any questions. Many customers called because they were confused. For example, many customers had this gripe: "I put my mouth next to the tape recorder, and yelled TWO PLUS TWO, but it didn't say FOUR!"

Radio Shack's first version of BASIC provided only three error messages: WHAT (which means "I don't know what you're talking about"), HOW (which means "I don't know how to handle a number that large") and OM (which means "I'm out of memory"). Those error messages confused beginners.

The following conversation actually occurred, between a Radio Shack customer (with a slow Texas drawl) and a Radio Shack technician who answered the 800 number (Chris Daly). . . .

Chris: "What's your problem?"

Customer: "Well, I plugged in the video, and then I plugged in the tape recorder, and then I . . . "

Chris: "Yes, sir, but what's the problem?"

Customer: "It doesn't work."

Chris: "How do you *know* it doesn't work?"

Customer: "It says READY."

Chris: "What's wrong with that? It's *supposed* to say READY."

Customer: "It *isn't* ready."

Chris: "How do you *know* it isn't ready?"

Customer: "Well, I asked it WHERE'S MY WIFE MARTHA, and it just said WHAT."

RADIO SHACK TODAY

Since those early days, Radio Shack has invented many improved versions of its TRS-80. Today, the TRS-80 comes in *eleven* versions.

The most elaborate versions are the Pocket Computer PC-2, the Color Computer 2, the Model 100, the Model 4P, the Tandy 2000, and the Model 16B. The stripped-down versions are the PC-4 & PC-3 (which are stripped-down versions of the PC-2), the Micro Color Computer (which is a stripped-down version of the Color Computer 2), the Model 4 (which is an old-fashioned version of the Model 4P), and the Model 12 (which is a stripped-down version of the Model 16B).

Let's look at each of those versions. I'll give you their list prices, rounded to the nearest dollar. (But you can get a discount of between 15% and 20%, if you buy from a discount dealer, such as Computer Plus at 617-486-3193 or 800-343-8124.)

POCKET COMPUTERS

The Pocket Computer PC-2 fits in the palm of your hand, runs on batteries, and includes an LCD screen. It looks like a pocket calculator, but its tiny keyboard includes all the letters of the alphabet (plus digits and symbols and function keys), and its ROM includes BASIC. It costs $200.

The PC-3 is smaller—it has a smaller keyboard, smaller screen, smaller RAM, and smaller ROM—and costs $100. The PC-4 is even smaller and costs just $70.

The PC-2 and PC-3 are manufactured for Radio Shack by Sharp. The PC-4 is manufactured for Radio Shack by Casio.

Because of their small screens and keyboards, all those Pocket Computers are awkward.

They're intended mainly for engineers who work outdoors and need a computer that fits in a pocket and runs on batteries. They compete against similar computers manufactured by Hewlett-Packard and Panasonic.

COLOR COMPUTERS

The Color Computer 2 attaches to your home's TV set.

It comes in three models. The $150 model includes an 8K ROM and 16K RAM. The $200 model includes a 16K ROM and 16K RAM. The $260 model includes a 16K ROM and 64K RAM.

The 8K ROM contains *Standard Color BASIC*. The 16K ROM contains *Extended Color BASIC*, which is far superior.

For $400 more, you can add a disk system. That price includes a disk drive, a 4-drive controller, a 4-drive cable, and a DOS. The disk system is single-sided double-density. Each additional disk drive costs $280.

The Color Computer 2 is intended mainly for students who want to learn programming.

Its main competitors are the Commodore 64 and Atari's new computers (the 600 XL and 800 XL). I prefer the Color Computer 2 over its competitors, because Extended Color BASIC is far superior to the versions of BASIC offered by Commodore and Atari, and because Radio Shack's DOS is much more convenient than the DOS's offered by Commodore and Atari, and because the Color Computer 2 is more reliable and durable than the computers made by Commodore and Atari.

Since Extended Color BASIC uses the same kind of wonderful commands for manipulating graphics and music as the IBM PC, you can consider it "a poor man's IBM". For students who can't afford an IBM PC or IBM PC Junior, I recommend the Radio Shack Color Computer 2 with Extended Color BASIC, wholeheartedly.

Radio Shack also makes a stripped-down version, called the Micro Color Computer MC-10, which costs just $80. But I advise against buying the MC-10, because of its tiny keyboard, its missing editor, and its stripped-down BASIC.

MODEL 100

The model 100 runs on batteries and fits in your lap. It includes a full-size keyboard and a large LCD screen (which shows eight 40-character lines). Its gigantic ROM (32K) includes a fancy operating system, a fancy version of BASIC, and four applications programs (for word processing, telecommunications, client addresses, and daily appointments). It includes a built-in modem, so you can attach it to a telephone.

To make the computer run on batteries, Radio Shack had to build the whole computer from low-power CMOS chips, which are expensive; so Radio Shack's price had to be high. The base price, $799, includes 8K of RAM; if you want 24K of RAM instead, you pay $999 total.

The model 100's main competitors are the Epson HX-20 and the NEC PC-8201A. The Epson HX-20 has the advantage of including a built-in printer and tape recorder, but suffers from a smaller screen and less software. The NEC PC-8201A is *very* similar to the model 100: its keyboard is slightly nicer and its RAM is slightly larger, though the quality of its software might be slightly worse.

DISK-BASED COMPUTERS

Radio Shack's main disk-based computers are the model 4P, the Tandy 2000, and the model 16B.

The model 4P costs $1799. It's a portable computer that contains a Z-80 CPU and two 5-inch floppy-disk drives, which are single-sided double-density. It competes against the Kaypro 2. Although the model 4P costs more than the Kaypro 2 and includes less software in its price, the *quality* of the software available for the model 4P is much higher than the quality of software available for the Kaypro 2.

The Tandy 2000 contains an 80186 CPU and runs the MS-DOS operating system. For $2750, you can get a Tandy 2000 that imitates an IBM PC and contains two 5-inch floppy-disk drives. For $4250, you can get a Tandy 2000 that imitates at IBM PC XT: it contains one 5-inch floppy-disk drive and one 10-megabyte hard-disk drive. The Tandy 2000 costs less than a comparably equipped IBM PC or IBM PC XT, and yet is *better* than an IBM PC or IBM PC XT in many ways: the Tandy 2000 includes a better CPU (80186 instead of 8088), better floppy-disk drives (double-sided *quadruple*-density!), better graphics, and a better keyboard. But because the Tandy 2000 is so new, not much software is available for it.

Radio Shack's most expensive computer is the model 16B, whose base price is $3999. That price includes a Z-80 CPU and also a 68000 CPU and an 8-inch floppy-disk drive. A second disk drive (floppy or hard) is optional. The model 16B is intended for doctors, lawyers, stockbrokers, accountants, and companies who want to network several computers together. Radio Shack sells lots of software for those applications. The model 16B competes against multi-user systems from Televideo and Altos.

The model 4 is an older version of the model 4P. Most people prefer the 4P, because the 4P costs less and is portable (the 4 is *not* portable). Radio Shack store managers expect that the model 4 will be phased out.

The model 12 is a stripped-down version of the model 16B: the model 12 has less RAM, only one CPU (the Z-80), and handles only one person at a time. Its base price is $2799.

PC JUNIOR

The PC Junior is manufactured for IBM by Teledyne, which also makes the Water Pik and a shower massage. Like other Teledyne products, the PC Junior is intended for good, clean family fun.

But cynics say that the PC Junior, like other Teledyne products, is "all wet", and are worried that Teledyne will try to replace the "light pen" by a "squirt pen".

The most controversial part of the PC Junior is the keyboard. Some typists think it's terrible; other typists think it's not so bad. Personally, I think it's not so bad, but my own feelings aren't important: what's important is how *you* feel. Walk into your local IBM store, tap on the PC Junior's keyboard for a half hour, and tell me how the keyboard feels to *you*.

The PC Junior itself costs $669. That price includes 64K of RAM, and it displays only 40 characters per line. For an additional $140, you get an additional 64K of RAM and the ability to display 80 characters per line. For an additional $480, you get a double-sided double-density disk drive (with its controller). Adding those three prices together ($669 + $140 + $480), you get a grand total of $1289. But if you buy all three at the same time, IBM will give you a $20 discount, so that you pay just $1269.

So suppose you walk into your local IBM store and pay $1269. You'll get a wonderful computer—that's totally useless!

It's useless because it lacks an essential ingredient: it lacks a screen. For a screen, IBM offers you a choice: either attach the computer to your home's TV (by getting a $30 RF modulator from IBM) or attach the computer to IBM's RGB color monitor (which costs $680, plus $20 for the cable).

Also, IBM's disk drive is totally useless—unless you buy IBM's DOS (version 2.1), which costs $65 extra.

Although the PC Junior includes a ROM that contains BASIC, the BASIC isn't very fancy. For a fancier version of BASIC, you must buy a ROM cartridge that costs $75 extra.

Here's a summary of all that information. The cheapest useable PC Junior consists of a $669 unit plus a $30 RF modulator, giving a total price of $699. A fully functional version of the PC Junior costs $1269, plus $30 for the RF modulator, plus $65 for the DOS, plus $75 for advanced BASIC, giving a total price of $1439.

If you want to attach the computer to your home's tape recorder, you must pay IBM $30 for the cable. To fully appreciate IBM's games and easy-to-use business software, you'll want IBM's joystick, which costs $40. To play a computer game with a friend, your friend will need a joystick too, which costs you another $40. Your total so far: $1439+$30+$40+40, which is $1549.

The keyboard runs on batteries, which are not included in the price! Also, the keyboard communicates with the main part of the computer via an infrared beam of light, which might be unreliable; for a more reliable connection, you'll want to buy a cord, which costs $20. You're now up to $1569 (plus batteries).

For a printer, you have a choice. For just $175, you can buy a printer that prints on thermal paper; that price includes the printer itself and all connections to the PC Junior. For a fancier printer, that prints on plain paper, IBM hits you with three expenses: $595 for the printer itself (a modified Epson MX-80), plus $55 for its cable, plus $99 for the Centronics parallel printer card that you put inside the PC Junior and that the cable attaches to; so the total cost for the fancy printer and connections is $595+$55+99, which is $749. Adding that $749 to the $1569 price for the PC Junior itself, you get a grand total of $2318.

The PC Junior is limited to 128K of RAM and 1 disk drive. If you want more RAM or a second disk drive, IBM requires you to buy the PC Senior instead.

PC SENIOR

The PC Senior (which is usually called just the "PC") costs more than the PC Junior, because it has a better keyboard, a larger power supply, more slots (for inserting more PC cards), and a larger motherboard that can hold more RAM.

If you want to buy a PC Senior, you face three expenses: the computer itself, the video, and the printer.

The computer itself. The base price of $1355 includes the 8088 CPU, the keyboard, a 40K ROM that contains BASIC, and 64K of RAM. To run all the popular software, you'll need to add $220 for disk controller card (which can handle two drives), $529 for a double-sided disk drive (make sure you get the double-sided version instead of the single-sided), $529 for a second double-sided disk drive, $60 for DOS version 2.0 (make sure you get version 2.0 or 2.1 instead of version 1.1, which is inferior), and $350 for an additional 192K of RAM (to bring the total RAM up to 256K). So your total cost is $1355 + $220 +$529 + $529 + $60 + $350, which is $3043. That's IBM's list price. Since IBM does *not* manufacture its own disk drives or RAM (it buys them from other suppliers), you can pay less by getting the disk drives and RAM directly from those suppliers or from independent dealers.

The video. For the video, you have four choices.

1. The fanciest choice is to buy IBM's RGB color monitor. You pay $680 for the monitor itself, plus $244 for the PC card it attaches to. That PC card is called the *color/graphics monitor card*: it contains the video interface chips, plus a ROM that handles graphics, plus 16K of RAM containing a copy of what's on the screen.

2. A less expensive choice—which is what I recommend for most people—is to buy a cheaper monitor from an independent company. You can buy color or black-and-white or green-screen, and pay as little as $99! But to attach such a monitor, you must still buy IBM's $244 color/graphics monitor card (even if your monitor is just black-and-white).

3. The cheapest choice is to use your home's TV. To do that, you must buy IBM's $244 color/graphics monitor card, plus a $70 RF modulator. But because your home's TV is slightly fuzzy, you won't be able to see 80 characters per line clearly; you'll want to use 40 fat characters per line instead. Since most of IBM's software insists on generating 80 thin characters per line, you won't be able to use most of IBM's software!

4. To get the most beautiful printing on a screen, get IBM's wonderful green-screen monitor ($345), which requires a special PC card ($335). Although that monitor prints characters more beautifully than any of the other three choices, it has two serious drawbacks: it cannot handle color and—even more important—*it cannot handle any graphics*. That's because its PC card lacks the ROM and RAM necessary for handling graphics.

The printer. To use a printer, you face three expenses: the printer itself, the printer's cable, and the printer's PC card. For the printer itself, you have many choices; instead of buying a printer from IBM (which uses Epson and NEC as its suppliers), you can save money by going to other dealers that sell Epson and NEC printers at a discount. For the printer's cable, IBM charges $55. For the printer's PC card, IBM charges $150; but you can omit that card if you chose video option #4 (whose $335 special PC card includes both a video interface and a Centronics interface).

Do you like the keyboard? As with the PC Junior, the most controversial part of the PC Senior is its keyboard. Although its keyboard feels nice, typists complain that its RETURN and SHIFT keys require a human's pinky to stretch too far.

Keytronics manufactures a keyboard that attaches to the PC Senior and has a more natural layout, so that your pinky doesn't have to stretch. Unfortunately, Keytronics' keys don't feel as nice as IBM's: Keytronics' keys feel too spongy.

Problems with imitations. You can buy an imitation of the entire computer, from companies such as Compaq, Columbia, Corona, Eagle, and Seequa. Most of those imitations use the Keytronics keyboard. Unfortunately, those imitations lack the 40K BASIC ROM; instead, the imitations require you to copy 40K BASIC from a disk to RAM. Also, some of the imitations lack the ROM and 16K RAM that IBM puts on its color/graphics monitor card. So on the imitations, you have less useable RAM than on the IBM PC. Also, the imitations use slightly different versions of DOS and of BASIC. Before you buy an imitation, make sure that the IBM software you want to use works on the imitation; ask the salesman to demonstrate it.

WHAT'S COLECO?

"Coleco" is an abbreviation for "Connecticut Leather Company".

The Connecticut Leather Company was started in 1932 by Maurice Greenberg, an immigrant from Russia. His company supplied shoe repair shops with scrap leather, heels, soles, and laces.

In 1944, his company began selling, directly to the public, a leather hat. The hat came in kit form: it was called the Howdy Doody Make-It-Yourself Bee-Nee. Then came Mickey Mouse Moccasins, Princess handbags, and Davy Crockett memorabilia.

Today, the company sells the Dukes of Hazzard Power Cycle, Smurf dolls, Cabbage Patch dolls, the Strawberry Shortcake Berry Buggy, Head-to-Head Football, above-ground swimming pools, pool tables, the Colecovision home-video-game system, and ROM cartridges that contain games for the Colecovision home-video-game system and also for the Atari VCS home-video-game system. Coleco also sells the Adam computer.

Because Coleco's products rely on fads (such as Howdy Doody, Davy Crockett, and video games), Coleco's fortunes are erratic. Although Coleco's sales steadily increase, its profits resemble a roller coaster:

Year	Coleco's sales	Coleco's net profit (or loss)
1973	$75,000,000	$2,400,000 profit
1974	$89,000,000	$1,000,000 profit
1975	$71,000,000	100,000 profit
1976	$117,000,000	$4,700,000 profit
1977	$137,000,000	$1,700,000 profit
1978	$107,000,000	($22,300,000 LOSS)
1979	$136,000,000	$5,400,000 profit
1980	$163,000,000	$13,100,000 profit
1981	$178,000,000	$7,700,000 profit
1982	$510,000,000	$44,900,000 profit
1983	$597,000,000	($7,400,000 LOSS)

1982 was highly profitable, because that was the year when Coleco started selling the Colecovision. During the first three quarters of 1983, Coleco made a profit of 27.6 million dollars, but those profits were wiped out by a disastrous fourth quarter, in which Coleco *lost* 35 million dollars. The loss was caused by difficulties in manufacturing the Adam computer.

Maurice Greenberg, the company's founder, died in 1980. Maurice's elder son, Leonard, is "chairman of the board", but devotes most of his time to charity instead of to Coleco. The younger son, Arnold, is Coleco's "president" and runs the Coleco's day-to-day operations. Since Arnold owns over a million shares of Coleco stock, and since Coleco's stock is currently selling for about $12 per share, Arnold is quite rich and also quite concerned about the stock market.

The company, whose name is now officially "Coleco Industries Inc.", is headquartered in an abandoned school building in West Hartford, Connecticut (203-725-6000). Coleco's Cabbage Patch dolls are built at a factory in Hong Kong; Coleco's Adam computer is built at an abandoned carpet mill in upstate New York, in a tiny town called "Amsterdam".

It's a small world! Coleco switched from leather goods to electronics to computers: so did Tandy! Coleco runs its computer operation from an abandoned textile mill: so does DEC!

THE ADAM ANNOUNCEMENT

Coleco invented the Adam secretly. Finally, at the Consumer Electronics Show in June 1983, Coleco announced the Adam publicly, and had an actual Adam on display.

The Adam was the hit of the show. All the newspapers and magazines gave it glowing praises. So did stockbrokers: Coleco's stock, which during the eight-year period from 1974 to 1981 had always stayed below $7 per share, and which during 1982 and the beginning of 1983 had occasionally nudged higher, suddenly shot up to $65 on the third day of the show.

Why was everybody so excited? The answer is simple: the Adam includes lots of hardware and software at an amazingly low price—and Coleco promised that even *more* amazing things would follow!

The Adam includes a wonderfully large keyboard: in addition to 54 standard keys, four arrow keys, and a HOME key, it also had 10 word-processing keys (INSERT, DELETE, BACKSPACE, CLEAR, PRINT, UNDO, WILDCARD, MOVE/COPY, STORE/GET, and ESCAPE/WP) and 6 programmable function keys (numbered I, II, III, IV, V, and VI), making a grand total of 75 keys. The keys had a good "feel", like a high-quality typewriter.

The Adam's game controller contains a joystick (a stick that you can pull in any direction, 360°), two fire buttons (which you press when you want to blow up the enemy), and a numeric keypad (which included twelve buttons, for the ten digits and for special operations). When you buy an Adam, you get (at no extra charge) *two* game controllers: so that altogether, you get two joysticks, four fire buttons, and 24 buttons on numeric keypads.

The Adam includes an RCA cord and a switchbox, which you attach to your home's color TV. The Adam also includes a jack for attaching to a color monitor.

The Adam's ROM contains a word-processing program that's easy to use, thorough, pretty (it puts color on your TV), cute (it even makes your TV draw a picture of a typewriter), and musical (it makes your TV play a happy musical note each time you hit a correct key, and makes your TV play a nastier sound whenever you goof; if your coworkers ever object to hearing such sounds, just turn down your TV's volume).

The Adam also includes a Z-80 CPU and a very large RAM (80K). It includes a very fast digital tape drive (almost as fast as a disk drive). When you buy an Adam, you get (at no extra charge) three tapes: the first tape contains an arcade game (Buck Rogers Planet of Zoom); the second tape contains an imitation of Apple BASIC; the third tape, which is almost blank, is used for storing your own programs and data files. Moreover, the Adam includes a slot into which you can insert any Colecovision ROM game cartridge.

The Adam also includes a bi-directional daisy-wheel printer that accepts friction-feed paper, so that it acts exactly like a typewriter. As you type on Adam's keyboard, the characters you type appear on the printer and the TV simultaneously. So even a "dumb typist" can use the Adam without any training: just turn on the power, start typing words on the keyboard, and as you type those same words appear on the printer's paper. You don't have to learn anything about word processing! That feature, which is called *electronic typewriter mode*, remains in effect until you hit the WP key (which does advanced word processing instead of ordinary typing) or until you play the BASIC tape (which does BASIC programming instead of typing).

So the Adam is an amazingly easy-to-use system. And at that Consumer Electronics Show, Coleco announced an amazingly low price: the Adam's list price would be just $600! That price included everything: the 75-key keyboard, the two game controllers with joysticks and fire buttons and numeric keypads, the TV hookup, the word-processing ROM, the Z-80 CPU, the 80K RAM, the digital tape drive, the three tapes (the Buck Rogers tape, imitation Apple BASIC tape, and data tape), and even the daisy-wheel printer.

At that time, the typical daisy-wheel printer in the marketplace was selling for over $600. Yet Adam's $600 price *included* a daisy-wheel printer. How could Coleco sell an entire computer system, including even a daisy-wheel printer, for less than the price of just a daisy-wheel printer? It didn't make sense. But Coleco insisted that it could and would sell the complete Adam system at that low price.

Coleco also published announcements and catalogs that listed many additional products for the Adam, such as a second tape drive, an additional 64K of RAM, CP/M, 5 business programs (electronic spreadsheet, spelling checker, phone directory, time manager, and home budgeter), 9 arcade-style games (Donkey Kong, Zaxxon, etc.), 5 adventure games (Cranston Manor, etc.), 10 educational packages (LOGO, typing tutor, Dr. Seuss, Smurfs, etc.), and a Z-80 macro assembler (for advanced programming). Less formally, Coleco also hinted at a disk drive and an 80-column card.

Coleco said that the Adam would arrive at stores by August.

THE PRICE WAR

When Coleco announced the Adam, Coleco's competitors were skeptical but also scared. They didn't believe that Coleco would be able to deliver such a wonderful product at such a low price by August; but they were also scared that Coleco *might* be able to do so. They were also scared because Coleco's Adam was suddenly becoming the focus of consumers' attention; nobody cared about Commodore or Atari anymore; everybody wanted to wait for the Adam to come out (and also wait for the IBM Peanut).

To retaliate, Commodore dropped the price of the Commodore 64 computer dramatically. Before the Adam was announced, the Commodore 64's list price had been $595, and discount dealers had started to move the price down to about $500. Immediately after the Adam announcement, Commodore dropped the 64's price to just $200.

When the 64's price dropped to $200, customers stopped buying Commodore's VIC-20 (which had been selling for $85) and switched to the 64. Suddenly, the VIC-20 became an obsolete computer. Companies who had written programs for the VIC-20 suddenly found themselves out of business.

Because the 64's price had dropped so low, people stopped buying the Texas Instruments 99/4A. Texas Instruments dropped the price and advertised heavily, but quickly gave up and left the home computer business.

DISAPPOINTMENTS

Although Coleco had announced a list price of $600, several mail-order houses began advertising the Adam at $675, because customer demand was so high and because the mail-order houses didn't really believe that the Adam would come out at a price as low as $600. The mail-order houses then began applying pressure on Coleco to raise the list price to $700.

Coleco promised that the Adams would arrive on store shelves at the end of August and that 500,000 Adams would be shipped during 1983. But August came and went without any shipments. Likewise, September came and went without any shipments. Coleco's stock dropped, and so did the stock of Toys R Us (which had planned to become the major source of Adams).

On October 18, Coleco shipped its first Adams to stores, so that they appeared on store shelves at the very end of October—two months behind schedule. The list price was $695, not $600. The wholesale price was $525. Toys R Us sold them to customers for about $550. About half of the Adams shipped were defective. Also, the version of BASIC and the manual that came with it turned out to both be buggy and incomplete. None of the promised extras were shipped: the "second tape drive, additional 64K of RAM, CP/M, 5 business programs, 9 arcade-style games, 5 adventure games, 10 educational packages (LOGO, typing tutor, Dr. Seuss, Smurfs, etc.), Z-80 macro assembler (for advanced programming), disk drive, and 80-column card" turned out to be just fictions invented by Coleco's ad department; the products didn't exist. Moreover, instead of shipping the 500,000 Adams that were promised, Coleco shipped just 95,000.

Although Coleco's Adam turned out to be a disappointment, Coleco's competitors were having problems of their own. Texas Instruments, after losing money, gave up trying to sell the TI 99/4A. The power transformer inside the Commodore 64 turned out to be often defective, and so did Commodore's disk drive. Atari's Hong Kong factory had trouble manufacturing the Atari 600 XL computer and didn't ship any. IBM's Peanut was announced in November and called the "PC Junior", but critics complained that it was overpriced and had a lousy keyboard, and IBM announced it wouldn't ship the PC Junior to customers until January and then announced that it had made a mistake on the date and wouldn't ship until March.

Because Coleco's competitors were having difficulties, Coleco raised the Adam's wholesale price $125 on January 1, 1984. So the wholesale price, which had been $525, went up to $650. To cover the $125 increase and still make a profit, stores had to raise the advertised price by $150.

Simultaneously, Atari raised its own prices also. The price war had ended.

ADAM'S MANUALS

While the Adam was being invented, Coleco signed a contract with the "Robert J. Brady Company", which is a publisher owned by Prentice Hall. The contract said the Brady bunch would write and print all of Adam's manuals.

Brady hired authors to write the manuals and showed the manuals to Coleco. Coleco approved most of the manuals but not the one about BASIC, since the BASIC manual was full of errors. Rather than correct all the errors, Coleco figured it would be easier to find a better author and tell him to start from scratch.

The Adam was invented by Eric Bromley, whose wife was one of my students. She showed him volume 1 of the old tenth edition of The Guide. He passed it on to Harvey Johnson (the marketing VP in charge of manuals), who showed it to his staff, which liked it because it was easy and fun. They wanted it to become the basis for Adam's manual.

I told Harvey that Coleco could use my Guide and that my coworkers would be glad to customize it for the Adam, although I'd be unable to oversee the customizing personally. I also recommended to Coleco and Brady that they also contact David Lien and Lon Poole, who are two top-notch authors who've written many famous computer books.

Harvey decided to use my Guide instead of hiring Lien or Poole, because Coleco's employees liked my writing style better. But instead of using my coworkers to customize the Guide, he said Coleco would use its own employees to do the customizing.

Coleco's main writer was Linda Lightfoot. Harvey told her to revise my Guide so that it would be specifically about the Adam and so that the sexual references would be acceptable "to Mormons in Utah". Using Utah as her standard, she made several clever changes. For example, she changed an example about "girls who meet boys" to an example about "gulls who meet buoys".

Unfortunately, because Coleco told Linda to rush, her manual was incomplete and contained many errors.

Linda's job would have been much easier, and Coleco's manual would have been much better, if Linda had received a huge pile of documents that I tried to send her. I gave to Harvey, to give to her, the entire 696-page text of the tenth edition, plus all relevant parts of the new improved eleventh edition, which I had customized for the Apple and which almost perfectly fit the Adam. But Harvey never gave those documents to Linda; he was afraid that so much information would confuse her. Instead, Harvey gave her just 96 pages of the obsolete tenth edition.

When Linda's deadline approached, and Linda suddenly found out from me that I had tried to send her the pile of documents, and she realized that most of the work she had slaved on for the entire month would have been eliminated if she had received that pile, she was furious! But it was too late: her crude expansion of the obsolete 96-page booklet became Adam's manual. As customers quickly found out, many of her hastily-constructed examples didn't work.

Because of Coleco's contract with Brady, I had to work through Brady. The final contract with Brady said that Brady and Coleco could use the 96 pages (which is all that Harvey wanted) at hardly any cost, under the condition that Coleco put into the Adam BASIC manual my name, address, phone number, copyright notice, disclaimer (saying that I was not responsible for Linda's errors and omissions), and a paragraph saying how the reader could obtain the new and complete eleventh edition of the Guide from me.

PHONE CALLS

Coleco's manual said:

"Should you find yourself in need of our assistance, our toll-free HELP hotline is 1-800-842-1225 nationwide, from 8:00 am to 5:00 pm Eastern Standard Time, Monday through Friday. Or, if our line is busy, you can reach Russ Walter at (617) 266-8128 if you have programming questions. You can call him 24 hours a day: he's almost always in, and he sleeps only lightly."

I received calls not only from Coleco's customers but also from Coleco's employees, who had been inadequately trained. The phone number made everybody happy.

But I soon became distressed at the pattern of the calls I had been receiving. The typical programming question that customers were asking me was: "How come *none* of the programs work?" I soon became aware that a large percentage of the Adams shipped were defective.

I became even more distressed when I began receiving calls from people who had gotten together with friends to buy large numbers of Adams, and found that *over 50%* of the Adams in the shipment didn't work.

I became even more distressed when customers told me that Coleco's employees weren't answering Coleco's toll-free number. Even when customers let the phone ring 30 times, still no employee answered the phone. It was as if Coleco had gone out of business. Since the toll-free number apparently wasn't working, I told those customers how to reach Coleco's main switchboard; but the main switchboard refused to help the customers and told them to just "keep trying" the toll-free number. The customers were quite upset, especially since many of them were calling to report that their Adams were defective.

The few customers who were lucky enough to get through to Coleco were told to mail their defective Adams to Coleco, for repair or replacement. The customers dutifully mailed their defective Adams to Coleco, received replacements, and discovered—much to their surprise—that the replacements didn't work any better than the originals! So the customers returned the replacements and received *second* replacements—which didn't work either! Several customers phoned me and told me that they were on their fourth or fifth replacements, which still weren't working!

During that time, when people asked me technical questions about the Adam's peculiarities, I had to tell them to ask Coleco's technicians instead of me, because Coleco hadn't sent me an Adam, and because Adams were not yet being sold in my part of the country. Imagine my surprise when, two weeks later, Coleco's technicians phoned me and asked me those same questions! I quickly discovered that Coleco's "technicians" knew as little about the Adam as I did.

The problems I've been describing are typical for any start-up company; but I was surprised to find such problems emanating from a company that had been in business for as many years as Coleco.

I chatted with several of Coleco's technicians about the problems I had seen, but they didn't seem to care. I tried to inform Coleco's management of the problems, but I was unable to get through.

DISPUTE

What finally occurred was a comedy of errors, which is too long for me to go into here. But the upshot of it all was: many reporters wrote negative articles about the Adam, in which they quoted customers, store owners, and me; Coleco replied by claiming that all Adam's problems had recently been corrected (which they had *not* been!) and also hired employees to rip my name out of Adam's manual, without telling me. Coleco and I are trying to settle this dispute; if you're curious about the outcome, phone me.

CONCLUSION

At the end of 1983 and during January and February of 1984, Coleco made a lot of improvements. Coleco significantly improved its BASIC manual, its word-processing program (which is in the ROM), its phone support for customers, and its repair service. Although further improvements should be made in all those areas, the Adam today is a fine product.

I have only a few complaints.

First, Coleco hasn't yet improved Adam's BASIC (which was designed for Coleco by Lazer Microsystems in California). The BASIC has three main bugs (involving decimals, random numbers, and high-resolution graphics) and can't yet handle music or high-speed animation.

Second, the information provided to the press and to customers by Coleco's management and customer-service operators is often incomplete or misleading or just plain false.

Coleco's word processor, though easy to learn how to use and cute, doesn't do much. It can't center a title; it can't make the right margin straight; it can't handle mailing lists; and it can't handle long documents well. It's also slow: it requires many keystrokes to perform even the simplest kind of editing; it moves the screen's text up and down too slowly; and it works only with Coleco's super-slow printer.

The power supply for the entire Adam computer system is inside the printer. If you send the printer in for repair, the rest of your Adam is without power and is useless. No other popular computer has that problem.

The tape that contains BASIC cannot be copied, and you can accidentally erase it too easily (by flipping the power switch at the wrong time). Since kids will accidentally erase that tape, the Adam won't survive in a school.

But Coleco's *main* problem is its reputation. Because the first Adams shipped were defective and because Coleco made so many statements that turned out to be false, many people are still afraid to buy the Adam, and many software companies are afraid that Coleco might go bankrupt. Therefore, the major software companies aren't developing any software for the Adam yet. (The software-company list that Coleco has been distributing is false.)

But maybe all those problems will be solved soon. Wait and see.

FOR ADAM OWNERS

To help Adam owners, I'm printing here the information that Coleco *should* have printed in its manual but didn't. Although I don't work for or with Coleco, the following seems to be the truth. . . .

Phone numbers. If you have a question about your Adam—or if your Adam seems broken and you want it repaired—Coleco's manual says to phone 800-842-1225 toll-free, and says that Canadians should phone 800-361-2122.

Unfortunately, those phone numbers are almost always busy. Keep trying. You can avoid a busy signal by dialing 203-521-7222 instead, but it's *not* toll-free.

When you finally get through (by dialing the toll-free number repeatedly, or by dialing 203-521-7222), you'll hear a voice and then hear music and then be put on hold for a long time. Eventually, somebody will help you.

If your question is very technical, you probably won't get an answer, but you stand a *slightly* better chance by dialing 203-521-7222 than by dialing the toll-free number.

Phone hours. The hours for 203-521-7222 are from 8AM to 5PM weekdays. For the toll-free number, the hours had been 8AM to 5PM weekdays, but they've been extended to 10PM.

Repairs. If your Adam needs repair, phone Coleco at one of those numbers. After approving your request, Coleco will tell you to mail your Adam to Honeywell, which will repair your computer (or get you a replacement) and mail it back to you. The service is free, and Coleco will even reimburse you for the postage. If, for any reason, your second computer goes bad, call Coleco again, and you'll get a third.

Supplies. To get supplies for your Adam, contact Coleco. Blank formatted tapes ("digital data packs") cost $6, printer ribbons cost $5.95, and daisy wheels cost $4.95. The only kind of daisy wheel in stock is the kind that came with your computer ("10 Pica"), but Coleco expects to stock two additional fonts soon ("Courier 72" and "Emphasis").

To order, send a check to Coleco Industries Inc., Consumer Services, 23 Chelton Ave., W. Hartford, CT 06110. If you live in Connecticut, New York, Illinois, or California, you must add sales tax. You must also add $1.25 to your total order, for postage and handling. To make sure you get what you expect, ask for the "Adam digital data pack, part #2564" and the "Adam printer ribbon, part #7806". Expect to wait about a month until you get what you ordered.

Coleco plans to sell a starter package that contains a tape, a ribbon, 3 daisy wheels, and an applicator. For the latest details about supplies, phone Coleco at the numbers I listed above.

Adam uses the same daisy wheels as other printers. To get a wide variety of daisy wheels for your Adam, go to your local computer store and ask for "Diablo print wheel #38101" or "Burrough print wheel #38101" or "National print wheel #97631".

New manual. Coleco's BASIC manual has gone through three major versions. The original version, which was shipped in October 1983, contained many errors. The second version, which was shipped in November and December 1983, included a blue page full of corrections. The third major version, which was shipped beginning in January 1984, corrected the corrections, didn't need a blue page, included extra chapters, and had a cover that said "revised edition" and was white instead of silver.

If you bought your Adam before January and therefore got an early version of the manual, phone Coleco (at one of the phone numbers I listed above): give Coleco your name and address. Coleco will promise to mail you the new version free . . . although Coleco has been slow at fulfilling that promise.

The purpose of the tapes. Your Adam comes with three tapes. The first tape contains a game: Buck Rogers Planet of Zoom. The second tape contains BASIC. The third tape (which I'll call the *data tape*) is mostly blank; use it for storing your own programs and files.

Caring for the tapes. The Adam's tapes are delicate. The TV and printer generate electromagnetic fields that garble the information on the tapes. Therefore, do *not* put the tapes on top of the TV or printer.

When you flick the Adam's power switch (from "off" to "on", or from "on" to "off"), the sudden surge of electrical power creates an electromagnetic field that garbles the information on your tape. Therefore, *remove all tapes from the drives, before you flick the power switch.*

For example, when you've finished using the Adam, do *not* just turn off the power: you should remove the tapes *first*, and *then* turn off the power. (Versions 1 and 2 of Adam's manual said the opposite but were wrong!)

How to begin using BASIC. To use BASIC, make sure no tapes are in the drive; then flick on the Adam's power switch. Then put the BASIC tape into the drive, and press the computer reset button. The tape will spin quickly, then slowly, and finally you should see the word BASIC at the top of the screen. (If your screen shows an electronic typewriter instead of the word BASIC, your BASIC tape is defective and must be returned to Coleco for replacement.)

How to test BASIC. To make sure BASIC is working properly, type this:

```
PRINT 345+2
```

When you press the RETURN key at the end of that line, the computer should say:

```
347
```

(If the screen shows 47 or 7 instead of 347, the 3 is left of your screen's left edge: move the 3 farther to the right, by turning your TV's HORIZONTAL HOLD dial; if the HORIZONTAL HOLD dial doesn't work or is missing, try a different TV. If the second TV doesn't work either, try a different Adam.)

Adam versus Apple. The Adam tries to imitate an Apple. So to learn how to program the Adam, read what The Secret Guide to Computers says about the Apple.

In a few small ways, the Adam differs from an Apple. Here's how. . . .

The Adam has less usable RAM than the Apple. Although the Adam contains 80K of RAM, most of that RAM is consumed by "system overhead", so that you have only 26K left for your program. (The Apple has 47K left instead.)

The Adam has trouble handling ".00". For example, if you type 3.00, the Adam will assume (incorrectly) that you mean a number slightly less than 3. If you type—

```
PRINT 3-3.00
```

the Adam will print:

```
9.31322574E-10
```

All other computers would print 0.

Since the Adam thinks that 3.00 is slightly less than 3, if you say—

```
PRINT INT(3.00)
```

the Adam will print 2. All other computers would print 3.

The Adam has trouble handling numbers such as 3.00, 4.00, and 5.00. In fact, it has trouble handling every such number from 1.00 to 1024.00. (But it can handle 1025.00 correctly.)

Although the Adam has trouble handling the ".00", it has no trouble handling ".0" or ".000".

The Apple permits each statement in your program to be long: up to 255 characters. The Adam requires that each statement be short: no more than 127 characters.

Like the Apple, the Adam uses the commands PR#1 and PR#0 to handle the printer. (For details, read volume 1's appendix about the Apple.) The Adam also provides an easier way of handling the printer: if you tap the P key while holding down the CONTROL key, the printer will print a photocopy of what's on the screen.

To clear the Adam's screen, type the word HOME and then press the RETURN key. Or, if you prefer, tap the L key while holding down the CONTROL key.

Do *not* put a space between GO and TO: the space confuses the Adam. Say "GOTO", not "GO TO".

To abort, hold down the CONTROL key; and while you keep holding down the CONTROL key, tap the C key. That aborts—usually. But if you try to abort while the Adam is doing an INPUT statement, you must press the RETURN key after pressing the CONTROL C.

You can store programs on Adam's data tape. (Do *not* store programs on Adam's BASIC tape.) The commands for handling Adam's blank tape resemble the commands for handling Apple's disk. For example, to put a program onto the tape and name it JOE, say SAVE JOE. But if you later revise JOE and put the revised version of JOE onto the tape (by saying SAVE JOE again), your tape will contain *two* versions of JOE: the revised version is called "A JOE"; the previous version is called "a JOE". If you say LOAD JOE or RUN JOE or DELETE JOE, Adam assumes you're talking about the revised version ("A JOE"). To use the previous version instead of the revised version, say DELETE JOE (which deletes the revised version) and then say RECOVER JOE (which changes "a JOE" to "A JOE").

To edit your program, use the arrow keys instead of the ESCape key. For example, to edit line 20, say LIST 20, then go the beginning of line 20 (by pressing the arrow keys), then hit the right-arrow until you reach the error you want to edit, then correct the error (to insert an extra space, tap the N key while holding down the CONTROL key; to delete a character, tap the O key while holding down the CONTROL key); then tap the right-arrow, until you get *past* the end of the line; then press the RETURN key twice.

The Apple's screen shows 40 characters per line. Adam's word processor shows 36 characters per line; Adam's version of BASIC shows just 31 characters per line. If you try to TAB beyond the 31st position, you'll confuse Adam—especially if you try using the printer.

Apple divides the screen into 3 zones. Adam divides the screen into just 2 zones.

Adam has trouble handling random numbers. If you run a program that contains random numbers and then run the same program again, you'll get the same "random" numbers as before.

If you draw a horizontal line (by using HPLOT) and then try to change the color of one of the points on the line, seven nearby points on the line will accidentally change color also.

While you're doing graphics, INVERSE and FLASH don't work. After doing graphics, if you say TEXT, Adam will automatically blank the screen.

To handle music and fancy graphics, the Apple uses the words PEEK, POKE, and CALL—and so does Adam. After the words PEEK, POKE, and CALL, you must put special numbers; unfortunately, Adam's numbers differ from the Apple's, and Coleco hasn't yet published Adam's numbers, so the PEEK, POKE, and CALL statements are useless.

Although Adam's tape drive can create "random-access files", it creates them *very slowly*, as if the computer were broken.

In the middle of your program, you can tell the computer to stop running the program and to start running program JOE instead. On both the Apple and the Adam, you do so by typing this:

```
40 D$=CHR$(4): ?D$"RUN JOE"
```

(Versions 1 and 2 of Adam's manual said to use the word CHAIN instead of RUN, but those versions were wrong. Adam does *not* understand the word CHAIN.)

If your data tape ever acts strangely, it's probably because some of the information on it got wrecked by an electromagnetic field. The only way to use that tape again is to erase it completely. To erase the tape, type this:

```
INIT HELLO
```

That command erases everything on the tape, and reformats the tape so that it acts properly.

ATARI

WHY ATARI?

In 1979, Atari began selling personal computers. At that time, Atari's computers were the most pleasant low-cost computers ever invented. In 1980, 1981, and 1982, Atari didn't sink enough effort into research, and so Atari's competitors began to catch up. But recently, Atari has introduced some new products that will make Atari "tops" again!

HOW ATARI BEGAN

Atari Incorporated was founded in 1972 by a guy named Nolan Bushnell. Atari's first product was a game called "Pong", which Nolan invented. It was the world's first popular video game. It played ping-pong. To play it, you had to insert quarters.

Nolan lived in California. He named his company "Atari", because "Atari" is an exciting Japanese war cry. (No, Nolan is *not* Japanese!) The word "Atari" is used in a Japanese version of chess; "Atari" means "check!" So it means, "Beware!"

After Pong, Atari's next successful arcade game was "Asteroids". Then came dozens of others. To play all those games, you had to insert quarters; the games were placed in arcades (and in your neighborhood bar).

In 1975, Atari invented a machine that played Pong on your home TV.

In 1976, Warner Communications Inc. bought Atari; and so Atari became a wholly-owned subsidiary of Warner. Warner is a gigantic company: it's owns DC Comics, Superman, Mad Magazine, Lauren perfumes, Warner Brothers movies and cartoons and television, Warner Cable TV, Atlantic Records, Elektra/Asylum Records, Knickerbocker Toy, and many other subsidiaries also!

After becoming part of Warner, in 1977 Atari invented the *Video Computer System (VCS)*, which is a machine that plays a wide variety of games on your home TV. Each game comes as a ROM cartridge. Later, companies such as Mattel and Coleco invented machines that were similar but fancier.

ATARI 400 & 800

In 1979, Atari began selling complete personal computers. Atari's first two computers were called the *Atari 400* and the *Atari 800*.

The Atari 400 used the same ROM and software as the Atari 800. The main difference between the 400 and the 800 was that the 400's keyboard was a monopanel (a single sheet of plastic, like the TS 1000), whereas the 800 had real moving keys (and was more expensive).

When Atari announced its computers, it said the Atari 400 would list for $550, and the Atari 800 would list for $1000. Those prices were amazingly low. At that time, Texas Instruments was still charging $1150, and Apple was charging even more. In later years, prices for Atari's computers (and its competitors) gradually declined.

The 400 and the 800 were far ahead of their time: they were wonderful and amazing!

Their color graphics were much more beautiful, much easier to program, and much more flexible than any other personal computer ever invented. (Compared to Atari, Apple's graphics are pitiful!)

Atari's computers produced four-part harmony through your TV's speaker. The music was pleasant (like a clarinet). Moreover, you could make Atari's computers produce outer-space sound effects. Programming your Atari computer to produce four-part harmony and sound effects was *easy*—you do it by writing just a short program in BASIC! (By contrast, the Apple could *not* produce harmony, and the only way to make the Apple play even a simple melody was to write a program in assembly language, which was much harder than BASIC.)

If you ever made a mistake while typing a program, your Atari would notice the error *immediately* and would point to the exact place in the line where the error occurred. (Most other computers don't notice errors until you type RUN, and they do *not* tell you where in the line the error occurred.)

If you made a error while typing a program, you could fix it by using Atari's built-in editor, which was much fancier and easier to use than any other computer's. (Even today, Atari's editor remains one of the best program editors ever invented. It's better than every program editor invented by Sinclair, Timex, Commodore, Texas Instruments, Radio Shack, Apple, Osborne, and Kaypro. The only editor that beats Atari's is IBM's, which is new and much more expensive.)

Atari's computers were safer than other computers, such as the Apple. If you tried using an Apple, you might create snow on your neighbor's TV; if you tried to open the Apple when the power was still on, you could burn your fingers and destroy your Apple; if you tried to insert a new chip into your Apple, you could accidentally break the caterpillar's delicate legs (pins). The Atari was idiot-proof: to prevent snow on your neighbor's TV, the Atari's case was made of thick radiation-proof metal (which was coated with cute plastic to look pretty); if you were stupid enough to open the Atari when the power was still on, the Atari automatically shut itself off; each extra chip that Atari sold was enclosed in a protective cartridge, so that you couldn't break the DIP's pins.

Atari offered exciting programs. They came on ROM cartridges.

The most popular ROM cartridge made the computer play a game called *Star Raiders*. The game featured high-speed 3D graphics that no other computer could handle.

At computer shows, Atari attached its computers to projection TV's, so that you could play Star Raiders on a gigantic screen. All the kids rushed to Atari's booth to play Star Raiders. The kids found it much more exciting than anything Apple had to offer. The kids told their parents to buy Atari instead of Apple.

Besides Star Raiders, Atari offered high-quality versions of other games (such as basketball). It also offered superior programs for education.

Atari's best education program was *Music Composer*. You can still buy it today. Music Composer makes your TV display a treble clef and bass clef, lets you easily draw notes, edit the notes, hear the notes (through your TV's speaker), create four-part harmony, and store your compositions on tape or disk. The program comes on a low-cost ROM cartridge. It's far better—and far less expensive—than the music programs offered by the other computer companies.

Although Atari's computers were wonderful, they weren't perfect. Let's look at Atari's mistakes. . . .

Atari's biggest mistake was its version of BASIC. Instead of hiring Bill Gates (Microsoft), Atari hired a different guy to create Atari BASIC. It was the same guy who had written Apple's disk operating system (Apple DOS).

Like Apple DOS, Atari BASIC seems easy the first day you use it. But on the second day, when you try to use it for advanced work, Atari BASIC (like Apple DOS) becomes noticeably hideous. The most hideous problem with Atari BASIC is the poor way in which it handles strings (words). Each time you want to use a string variable, you must warn the computer how many characters will be in the string: to do that, you must put the word DIM in your program; after the word DIM, you must put the number of characters that will be in the string. That restriction prevents you from easily creating a list of strings. Therefore, you can't easily make the Atari alphabetize a mailing list. Also, Atari's BASIC handles disks poorly.

Too late, Atari realized it should have hired Bill Gates (Microsoft) to create Atari BASIC. Atari fired the manager who had made the mistake of not hiring Bill.

Then Atari tried to hire Bill to improve Atari's BASIC. But Bill raised his price from $50,000 to $75,000. Atari balked at paying $75,000. Eventually, Atari agreed to pay the $75,000. But it was too late. Atari's original BASIC had gotten such a bad reputation that programmers avoided Atari's computers like the plague. Hardly anybody was writing programs for Atari's computers. Atari's percentage of the home computer market declined.

When Atari *did* finally offer Microsoft BASIC, the result was disappointing. Atari's original BASIC came in a ROM cartridge, but Microsoft BASIC came only on a disk. Since the average Atari owner was too broke to afford a disk drive, the average Atari owner couldn't use Microsoft BASIC and had to use Atari's original crummy BASIC instead.

But even if you were rich enough to buy a disk drive and Microsoft BASIC, your problems weren't over. Since Atari feared you'd give unauthorized copies of Microsoft BASIC to your friends, Atari made the disk uncopyable. To make the disk uncopyable, Atari unfortunately used a protection scheme that was awkward: not only did it make copying difficult, but it also hindered your use of the original disk you had paid for. That hindrance prevented Atari's Microsoft BASIC from becoming popular.

In maxicomputers, minicomputers, and most microcomputers, each PC card has an *edge connector*, which is on the edge of the PC card and connects the PC card to the motherboard; the edge connector is made of gold. To reduce costs, Atari used tin instead of gold. Unlike gold, tin oxidizes. So if you bought an Atari computer, you had to open the computer frequently and wipe off the tin's oxidation, by using a rag or a rubber eraser. If you did *not* wipe off the oxidation, Atari computer acted unreliably. Since most consumers didn't know they had to wipe off the tin, consumers complained that the computer was unreliable. (After several years of complaints, Atari finally gave in, and began using gold.)

Another problem was overheating, because the entire computer was enclosed in a thick metal case that was poorly ventilated. As a result, your TV screen would suddenly turn a sickening orange-pink, the keyboard froze, and you had to wait for the computer to cool down.

Also, Atari's tape recorder was unreliable.

Although most programmers use BASIC, some programmers prefer a more "sophisticated" language called *PASCAL*. In 1979, when Atari began selling computers, Atari said its computers would understand PASCAL by the end of the year. Programmers waited for PASCAL . . . and waited . . . and waited. They waited for many years. They're still waiting. Even today, Atari computers don't understand PASCAL.

Atari didn't worry about its awkward BASIC and its lack of PASCAL, because Atari didn't believe most purchasers of home computers would want to write programs. Atari believed most purchasers would want to *buy* programs that played games or that were educational.

Atari was wrong: today most people who buy home computers *do* want to write programs. And Atari's poor BASIC has cost Atari many sales.

Atari didn't care much about its personal computers, because they were less profitable than Atari's other products: coin-operated arcade games and home video games. In fact, because Atari's personal computers generated so little income, Atari's personal-computer division actually ran in the red!

So Atari ignored its personal computers. Instead of personal computers, Atari sunk its efforts into Atari's more profitable products: it spent lots of money of inventing new arcade games and home video games and advertising them.

But in 1982, profits from Atari's arcade games and home-video games started to drop, because of increased competition from other vendors and because the American public started to tire of the video-game fad. Then Atari regretted that it hadn't put more effort into its personal computers.

At the end of 1982, Atari announced it would sell a new computer, called the *Atari 1200 XL*. (The "XL" stands for "excel".) It would be slightly better than the Atari 800: the Atari 1200 XL would be more reliable, have more keys on the keyboard, and handle more RAM. On the other hand, it would have a *dis*advantage: it wouldn't handle as many joysticks and printers as the 800. It would cost more than the 800; in fact, it would cost close to $1000.

We were all disappointed at the high price and at Atari's lack of creativity. It won a prize for being the most boring new computer. Clubs of Atari users hated it so much that they nicknamed it the "Edsel".

In 1983, Atari finally came to its senses and lowered the price. Atari also announced more new computers.

Theoretically, Atari's new computer lineup consists of the 600 XL, the 800 XL, the 1200 XL, the 1400 XL, and the 1450 XLD. Each of those computers includes a real keyboard (*not* just a sheet of plastic). Each of those computers contains a 24K ROM, which includes the operating system and Atari BASIC. Each of those computers contains a 64K RAM, except for the smallest (the 600 XL), which contains only 16K. The largest (the 1450 XLD) includes a double-sided double-density disk drive.

That's the theory. In practice, Atari has started shipping the 600 XL and 800 XL (which work well), has stopped shipping the 1200 XL, and has indefinitely postponed the 1400 XL and 1450 XLD.

Because of those XL computers, the original Atari 400 and Atari 800 are obsolete.

Atari has also invented a new tape recorder (model #1010), which is more reliable than Atari's previous one (model #410).

Finally, Atari has announced a satisfactory version of Microsoft BASIC. It's called *Atari Microsoft BASIC 2*. It lists for $89.95. For that price, you get a cartridge that contains a 16K ROM that makes the computer understand most of Microsoft BASIC. You also get a disk that contains 3½K of additional Microsoft BASIC. If you don't have a disk drive, don't worry: you can use the cartridge without the disk. You can copy the disk.

Atari's new version of Microsoft BASIC, its new reliable XL series, its new low prices, and its new tape recorder and disk drive have all restored my faith in Atari, and I recommend Atari computers very highly (if you also buy Atari Microsoft BASIC 2).

But I must admit that the company occasionally makes me nervous. For example, Atari's president called a meeting of all his top salespeople, so that they could all admire the new Atari 1200 XL and see how great it was. The Atari 1200 XL's product manager plugged in his new computer. A puff of smoke came out. The president, a master of understatement, said, "It looks like this computer isn't safe yet."

I hope you have better luck with *your* new Atari computer!

PROGRAMMABILITY

DARTMOUTH BASIC

BASIC comes in several different dialects. The original dialect was invented at Dartmouth College in 1964. It was primitive: it didn't even include the word INPUT (you had to say READ instead), and it didn't even include string variables. Afterwards, Dartmouth rapidly improved its version of BASIC, so that today Dartmouth sues what it calls the "7th edition".

Today, many manufacturers brag that their computers have "improvements over the original Dartmouth BASIC". That's not much to brag about, since even *Dartmouth* has improvements over the original Dartmouth BASIC.

Dartmouth's original research was done on a General Electric computer. Afterwards, General Electric quit making computers and sold its computer factory to Honeywell. So today, Dartmouth's computer is called a "Honeywell".

Since much of Dartmouth's research on BASIC was funded by the National Science Foundation, the results of the research were in the public domain, and computer manufacturers could use them without having to worry about copyrights or patents.

DEC BASIC

The first company to copy Dartmouth's ideas was Digital Equipment Corporation (DEC). DEC put BASIC on DEC's minicomputers (the PDP-8, PDP-11, and VAX-11) and DEC's maxicomputers (the Decsystem-10 and Decsystem-20). DEC's versions of BASIC were similar to Dartmouth's. The versions put on the PDP-8 were quite primitive (almost as bad as Dartmouth's first edition), but the versions put on DEC's other computers were more sophisticated. Eventually, DEC put decent versions of BASIC even on the PDP-8, so that today all of DEC's versions of BASIC are decent.

Today, DEC's best version of BASIC is *VAX BASIC*, which works only on the VAX-11 computer. DEC's second-best version of BASIC is *BASIC-PLUS-2*, which works only on the PDP-11, the VAX-11, and the Decsystem-20. DEC's third-best version of BASIC is *BASIC-PLUS*, which works only on the PDP-11. DEC's other versions of BASIC aren't as nice, though still acceptable.

DEC'S COMPETITORS

Soon after DEC started putting BASIC on its computers, Hewlett-Packard decided to do likewise: Hewlett-Packard put BASIC on the HP-2000 computer, and then put a better version of BASIC on the HP-300 computer.

Unfortunately, Hewlett-Packard's BASIC was more difficult to use than DEC's. On Hewlett-Packard computers, each time you used a string you had to write a "DIM statement" that warned the computer how long the string would be: the DIM statement had to say how many characters the string would contain.

Later, other manufacturers decided to use BASIC also. Most manufacturers imitated Honeywell and DEC; but Data General, Wang, and IBM made the mistake of copying Hewlett-Packard instead.

That's how BASIC developed on maxicomputers and minicomputers.

MICROSOFT

The first popular *micro*computer was the Altair 8800, which used a version of BASIC invented by a 20-year-old kid named Bill Gates. His version imitated DEC's.

The Altair computer was manufactured by a company called *MITS*. When MITS didn't treat Bill Gates fairly, he broke away from MITS and formed his own company, called *Microsoft*.

Bill Gates, and his company Microsoft, invented many versions of BASIC. The first was called *4K BASIC*, because it consumed only 4K of semiconductor memory (RAM or ROM). Then came *8K BASIC*, which included a larger vocabulary than 4K BASIC. Then came *Extended BASIC*, which consumed 14K. All those early versions were intended for computers that used tapes instead of disks. Finally came *Disk BASIC*, which came on a disk and included all the commands for handling disks. His *Disk BASIC version 4* was further improved, to form *Disk BASIC version 5*, which is the version of BASIC still used on CP/M computers and on the Radio Shack models 4 and 4P. It's also called *MBASIC* and *BASIC-80*.

All those versions of BASIC were written for computers that contained an 8080 or Z-80 CPU. Simultaneously, Bill and Microsoft wrote *6502 BASIC*, for Apple and Commodore computers. Apple's version of 6502 BASIC is called *Applesoft BASIC*; Commodore's version of 6502 BASIC is called *Commodore BASIC*.

Unfortunately, 6502 BASIC is rather primitive: it resembles Bill's 8K BASIC. So if you're trying to learn advanced BASIC programming, you should *not* choose an Apple or Commodore computer!

After writing 6502 BASIC, Bill wrote an improved version of it, called *6809 BASIC*, which is available only for Radio Shack's Color Computer. Radio Shack calls it *Extended Color BASIC*. It's approximately as powerful as Bill's *Disk BASIC version 4*.

Texas Instruments (TI) asked Bill to write a version of BASIC for TI computers. Bill said "yes". Then TI told Bill what kind of BASIC it wanted. Bill's company—Microsoft—found 90 ways in which TI's desires would contradict Microsoft's traditions. Microsoft convinced TI to change its mind and remove 80 of those 90 contradictions, but TI stood firm on the other 10. So today, TI BASIC (which is on the TI-990 and TI-99/4A computers) contradicts all other versions of Microsoft BASIC in 10 ways. For example, in TI BASIC, the INPUT statement uses a colon instead of a semicolon, and a multi-statement line uses a double colon (::) instead of a single colon. Because of those differences, TI's computers have become unpopular. Moral: if you contradict Bill, you'll become unpopular!

Bill's newest and best BASIC is called *GW BASIC* (which stands for "gee-whiz BASIC"). It requires an 8088 or 8086 CPU, and works only on the IBM PC and its clones.

PAY YOUR BILL

Some manufacturers tried to avoid paying Bill. They were sorry!

For example, RADIO SHACK tried hiring somebody else to write Radio Shack's BASIC. That person quit in the middle of the job; so Radio Shack's original BASIC was never completed. Nicknamed "Level 1 BASIC", it was a half-done mess. Radio Shack, like an obedient puppy dog, then went to Bill, who finally wrote a decent version of BASIC for Radio Shack; Bill's version was called "Level 2". Today, Radio Shack uses further improvements on Bill's Level 2 BASIC.

The same thing happened with APPLE. Apple's original attempt at BASIC was called "Apple Integer BASIC". It was written by Steve Wozniak and was terrible: it couldn't handle decimal points, and it made the mistake of imitating Hewlett Packard instead of DEC (because Steve had worked at Hewlett-Packard). Eventually, Steve wised up and hired Bill, who wrote Apple's better BASIC, called *Applesoft* (which means "Apple BASIC by Microsoft"). Applesoft was for use only with tape. Later, when Steve Wozniak wanted to use disks, he made the mistake of not rehiring Bill—which is why Apple's disk system is worse than Radio Shack's.

A computer manufacturer called "Kentucky Fried Computers", which changed its name to NORTH STAR COMPUTERS, didn't want to pay Bill. So it invented its own "North Star BASIC", which imitated Hewlett-Packard's. North Star BASIC was a disaster. Most people who still own North Star computers are buying the CP/M operating system, so they can use Microsoft's disk BASIC version 5 instead.

At ATARI, an executive didn't want to hire Bill, and instead hired the same guy who had invented Apple's disastrous DOS. That guy's BASIC, which is called *Atari BASIC*, resembles Hewlett-Packard's BASIC. It has the same characteristics as Apple's DOS: it looks pleasant at first glance, but turns into a nightmare when you try to do any advanced programming. As a result, Atari's computers never became as popular as Atari hoped, and the executive who "didn't want to hire Bill" was fired, and Atari has finally hired Bill's company, which has written *Atari Microsoft BASIC version 2*.

A small computer manufacturer, called APF, invented a computer called *The Imagination Machine*. It was beautiful, and its price was ridiculously low, and it would have been a smashing success, except for one mistake: APF didn't hire Bill. Instead, APF hired a jerk who wrote the worst version of BASIC that I've ever seen (many, many errors!). Result: APF is no longer in the computer business.

Of course, IBM didn't fool around: when IBM invented the IBM PC, it hired Bill immediately, as a consultant. IBM realized that Bill was better at developing BASIC than any of IBM's employees.

Moral: if you're planning to manufacture a microcomputer, make sure you pay Bill! And if you're trying to *buy* a microcomputer, make sure it handles Microsoft BASIC.

COMPARISON CHART

Of all the popular computers being sold today, which is the easiest to program? To find out, look at this chart:

	Logic			Program entry		Visuals		
	BASIC statements	string variables	INPUT prompt	keyboard	BASIC's editor	screen width	character set	graphics commands
Timex Sinclair								
Timex Sinclair 1000	1	3	1	1	2	1	1	1
Timex Sinclair 1500	1	3	1	2	2	1	1	1
Timex Sinclair 2068	3	3	3	2	2	1	3	3
Texas Instruments 99/4A								
without Extended BASIC cartridge	2	3	3	2	2	1	3	1
with Extended BASIC cartridge	3	3	3	2	2	1	3	1
Commodore VIC-20								
without Super-Expander cartridge	3	3	2	3	2	1	3	1
with Super-Expander cartridge	3	3	2	3	2	1	3	3
Commodore 64								
plain	3	3	2	3	2	2	3	1
with Super-Expander or Simon's	3	3	2	3	2	2	3	3
Atari								
Atari 400 with Atari BASIC	3	1	1	1	2	2	3	2
Atari 600 with Atari BASIC	3	1	1	3	2	2	3	2
Atari 600 with Microsoft BASIC	3	3	3	3	2	2	3	2
Radio Shack Color Computer 2								
with standard BASIC	3	2	3	3	1	1	1	1
with Extended BASIC	3	2	3	3	2	1	1	3
Radio Shack Model 4								
without disk	3	2	3	3	2	2	3	1
with disk	3	3	3	3	2	3	3	1
Mid-priced brands								
Apple 2e	3	3	3	3	1	2	2	2
Kaypro	3	3	3	3	2	3	2	1
IBM Personal Computer								
plain	3	3	3	3	3	3	3	1
with disks & color card	3	3	3	3	3	3	3	3

In the chart, 3 means "good", 2 means "fair", and 1 means "bad". If a computer gets many 3's, it means the computer is good for programming: in other words, it's easy to program. A computer that gets many 1's is awkward to program.

Of all the computer systems shown in the chart, the best is the "IBM Personal Computer with disks & color card". It's the only computer system that scores a perfect 3 on every test.

It's also the computer system that costs the most. It lists for $2218: that gets you the computer with 64K of RAM, a disk controller card, a single-sided disk drive, DOS version 1.1, an RF modulator, and a PC card to attach the RF modulator to. After spending the $2218, you'll also want to spend extra money also, to get extra RAM, a disk drive that's double-sided instead of single-sided, a second drive, and DOS version 2.0.

Of the computer systems shown in the chart, the worst is the "Timex Sinclair 1000". It scores a 1 (which means "bad") on almost every test. It's also the computer system that costs the least: it lists for just $49.95.

Let's look at the eight tests.

BASIC statements. If the computer understands the words READ and DATA and also understands colons without any hassles, it scores a 3. If it understands READ and DATA but doesn't understand the idea of putting a colon between statements, it scores a 2. If it doesn't understands READ and DATA, it scores a 1.

String variables. If the computer lets you use string variables without any hassles, it scores a 3. If you must say CLEAR in order to use long strings, the computer scores a 2. If you must say DIM in order to use string variables, the computer scores a 1.

INPUT prompt. If the computer lets you put a prompt in an INPUT statement without any hassles, it scores a 3. If you must make the prompt be short, the computer scores a 2. If the INPUT statement can't contain a prompt at all, the computer scores a 1.

Keyboard. If the keyboard is standard, it scores a 3. If the keys work in the standard way, but the keyboard's layout is awkward (because quotation marks and other special symbols are on the wrong keys), the keyboard scores a 2. If the keyboard has no moving keys—if instead the entire keyboard is just a single sheet of flabby plastic that you squeeze—the keyboard scores a 1.

BASIC's editor. If you can correct your program very easily, by using a built-in editor, the computer scores a 3. If the editor is slightly awkward to use, the computer scores a 2. If the editor is extremely awkward to use—or is missing altogether—the computer scores a 1.

Screen width. If the screen can display 80 characters per line, and the version of BASIC lets you manipulate all 80 characters easily, the computer scores a 3. If BASIC won't let you easily handle 80 characters, but will at least let you handle 36, the computer scores a 2. If BASIC won't even let you handle 36, the computer scores a 1.

Character set. Every computer can show capital letters on the screen. If you can easily write a BASIC program that makes the screen also show lower-case letters and graphics characters (such as little hearts and diamonds), the computer scores a 3. If BASIC can handle lower-case letters but can't handle graphics characters, the computer scores a 2. If BASIC can't make the screen show lower-case letters, the computer scores a 1.

Graphics commands. If BASIC includes commands that draws long diagonal lines and big circles, the computer scores a 3. If BASIC includes a command that draws long diagonal lines but doesn't include a command to draw big circles, the computer scores a 2. If BASIC doesn't include a command that draws long diagonal lines, the computer scores a 1.

STRANGE DEVICES

THE TALKING CAR

Back in the 1950's, a car's dashboard contained many dials. For example, one dial told you how much oil pressure you had; another dial measured your battery's charge and discharge.

But drivers were too stupid to figure out how to read the dials! So the car manufacturers replaced the dials by "idiot lights", which glow red if anything is "wrong". Any idiot can understand those lights: red glow means "bad", no glow means "okay".

But unfortunately, some drivers are so idiotic that they can't even remember to look at the idiot lights! Those drivers peer straight ahead at the road and never look at their dashboards. Those drivers are dangerous: they're daydreaming or asleep-at-the-wheel or drunk or high or just plain stupid. To help those drivers, now you can buy a car that talks! The voicebox hides inside the dashboard and is run by a type 1 computer. If your oil's getting low, the voicebox says, in a loud voice, "Oil low!" If you're going to run out of gas soon, the voicebox says "Need more gas!" That voicebox can help save your car and even your life—unless you get so scared by hearing the robot voice that you have a heart attack!

CASIO SYNTHESIZER

My favorite computerized device is the VL-1 Music Synthesizer, manufactured by Casio in Japan.

It's about the size of a foot-long ruler. It includes a piano-style keyboard. The keyboard includes the white notes and the black notes. The keyboard is small (only $2\frac{1}{2}$ octaves), but you can get 2 additional octaves by flicking a special switch that raises or lowers the pitch by an octave.

Another special switch lets you choose your tone quality: you can flick that switch to "piano", "violin", "flute", "guitar", "fantasy", or to "make up your own!" If you choose "make up your own", you then type a number from 0 to 99,999,999; the number you type is the instrument you get; so altogether, you can get 100,000,000 different musical instruments!

While you're playing one of those instruments, you also get a percussion section to back you up: you can choose 10 different kinds of rhythm: "march", "waltz", "swing", "samba", "rhumba", "bossanova", "beguine", "4-beat", "rock", and "fancy rock". For each of those rhythms, you can choose 19 different speeds; so altogether, you have 190 different combinations of rhythm and speed!

The machine includes a speaker, so that you can hear your composition.

But the best part of all is: the machine will memorize everything you play! That's because, while you're pushing the keys and buttons and listening to your music and having a good time, the computer that hides inside the machine is noticing your every move and recording it in the RAM. Whenever you wish, flick another switch and you get an instant replay of everything you did!

If you're a bad musician, you can press the keys slowly but then tell the computer to play the composition back faster, as if you were good. If you accidentally hit a wrong note, just press the "delete" key, and the note will go away: when you ask for the instant replay, the bad note will be gone, as if you were perfect!

The machine also acts as a calculator: if you press the key marked "2" and then press the key marked "square root", it will say "1.4142135", so that you can write the machine off as a business expense.

It comes in a lovely leatherette carry case. Batteries are included. You get instructions in both English and Spanish, and you also get a book of songs from all around the world.

The total price for all that? The list price is $80; during the Christmas of 1981, most American department stores sold it for $55; and it was selling for just $37.50 in New York City.

Only computer technology could produce such a wonderful machine at such a low price!

Casio and competitors (such as Yamaha) also make fancier versions (at slightly higher prices). Today, most stores sell only the larger, fancier versions. But that cheap little bugger sure was cute!

GAME MACHINES

Many companies sell tiny machines that fit in your pocketbook and play games. For example, you can buy a machine that plays chess and a machine that plays bridge. Inside each such machine hides a tiny microcomputer.

Such machines have been around for several years and are no longer considered unusual. But I was surprised to hear about a machine that fits in your lap and plays Scrabble against you by using its ROM, which contains all the rules of Scrabble plus all the important words from the Official Scrabble Player's Dictionary! The machine is called the *Monty Plays Scrabble Portable Computer Console*. It costs $149.50 for a version that includes 12,000 Scrabble words; add $32.50 for an additional 16,000 words; add $32.50 for a second bank of 16,000 words (bringing the total number of words it knows to 44,000). It's a good sport: for example, if you enter a high-scoring word, it won't hiss at you; instead, it prints "GOOD PLAY" on its LCD screen and plays a few bars of Tchaikovsky's 1812 Overture. If you love Scrabble but can't find any friends who want to play with you, consider buying that Scrabble computer: it bridges the gap between science and the humanities. Too bad it's so expensive: $149.50 + $32.50 + $32.50 = $214.50. But if you already own a Radio Shack model 3 or model 4 or model 4P computer with disk drives, you can get the same program for just $34.95 instead of $214.50! Yes, $34.95 is the price of Radio Shack's *Monty Plays Scrabble Disk*. The disk includes the entire Scrabble program plus a super-large dictionary: 54,000 words!

Problem: you must attend an important high-level meeting of executives, and you must give them the impression that you too are a high-level impatient executive. But actually the meeting bores you. What should you do? Solution: now you can buy a digital watch whose tiny screen also lets you play video games, such as Pac-Man and bowling. Throughout the boring meeting, frequently look at your watch (and play the game). When you look at your watch, make sure to frown. When you frown at your watch, the executives will think you're impatient with them and that you must therefore be even higher-level than they.

ご村広戸ず王見 2

BUSINESS APPLICATIONS

GROUNDWORK

HOW TO BUY A COMPUTER

First, decide *why* you want a computer. What do you want to use the computer for? Then decide how many people will want to use the computer simultaneously: if the answer is "many", you'll need to buy a computer that has many terminals, or you'll need to hook many single-user computers together, to form a network. Then estimate how large your files are: how many megabytes of disk will they require?

Then hunt for the lowest-cost computer that meets your needs: hunt for a computer that has enough megabytes, allows enough simultaneous users, and comes with adequate software and programs. Decide whether you can afford it: if not, don't computerize yet.

Then examine other computers that are slightly more expensive but slightly better, and decide whether their "extras" are useful enough to be worth paying for. Last but not least, decide whether to buy the computer from your local dealer (at list price) or mail-order (at discount).

Begin by looking at microcomputers. For most readers of this book, microcomputers are adequate. But if your needs are very large— if you need to let many people share many megabytes— you might have to shell out more money, and get a minicomputer or maxicomputer instead.

The most popular microcomputer manufacturers are Radio Shack and Apple. Begin by looking at their products, then consider the competition.

At first glance, having a computer sounds nifty. But if you get a computer, you'll have lots of hassles.

REPAIRS

Since a complete computer system contains so many parts (the CPU, RAM, keyboard, disk drives, printer, etc.), *at least one* of the parts will be broken, and you'll need to get it repaired. If you get the repair done promptly (within 90 days), the manufacturer or store will pay for the repair, so you've lost nothing... except your temper.

MANUALS

Another difficulty is the manual that comes with the computer. Chances are, you won't understand it, and you'll have to ask your friends for help, or hire a consultant, or call me. (Since I'm a public-service organization, I don't charge you anything for help I give over the phone.) Of course, you can also try getting help from the company that sold you the computer; but unfortunately, most companies tend to be somewhat curt, especially if your question is long-winded.

If the salesman is candid, he'll say, "I don't know how to run the computer I sold you. If you want to learn how to run it, read the manual. No, I haven't read the manual myself; it's too long-winded and complicated and vague. If you don't like the manual, go take one of our courses. The course is expensive and won't teach you as much as you need to know, but at least will make you feel you're making *some* progress."

Most salesmen are not that candid.

PROGRAMS

If you try to write your own programs, you'll discover Murphy's law: no matter how long you think the program will take to write, it will take you longer. If you're wiser, and try to buy a pre-made program from somebody else, you'll find that the program doesn't work as well as the ad implied, the program's manual is either missing or impossible to understand, and— more important — you'll need to change the program to make it handle your own personal needs.

DATA ENTRY

Suppose you're lucky, and finally figure out how to use the program. Your next odious chore is to feed in the data that you want the program to process. Feeding data into a computer program is sheer drudgery, but you must do it (or stick it to a minimum-wage employee).

COWORKERS

Your next unhappy surprise is that your coworkers will both hate and fear the computer. You must become their teacher and psychologist.

SUMMARY

Those hassles are only the tip of the iceberg, but the message is clear: buying a computer at first seems joyfully exciting but quickly becomes nerve-racking.

Finally, however, the nerve-racking transition stage will be over, and you'll be thrilled.

Question: is the nerve-racking transition stage worth the effort? If you're going to be using the computer frequently, the answer is: yes. If you'd be using the computer only infrequently, the answer is: don't bother buying a computer; continue doing your work manually.

MINIMUM REQUIREMENTS

Let's assume you've definitely decided that a computer would be useful enough to justify the effort spent on the transition. So you've decided to buy a computer, under the conditions that (1) the computer is powerful enough to fill your needs and (2) the price is low enough to be affordable.

Which computers are powerful enough to meet your needs? The answer depends on what you want to use a computer for.

If you want a computer to help run your business, buy a computer that has at least 2 disk drives, at least an 80-column impact printer, and at least 40K of main memory. (Most microcomputers satisfy the "40K" requirement by giving you 32K of RAM plus at least 8K of ROM.) Those are the *minimum* requirements for creating a decent business system.

Here's why you need at least 2 disk drives.... To be safe, you should always make a second copy of whatever you put on a disk— in case the first copy gets accidentally destroyed. The second copy should be on a second disk. You should store the second disk far away from the first disk, to reduce the chance of both disks getting destroyed simultaneously. Your computer should include a second disk drive, to let you produce the copy easily.

You need at least 40K of main memory, because the typical business program consumes about 16K, the typical disk operating system consumes about 8K, and the typical BASIC-interpreter-and-editor-and-monitor-and-fundamental-operating system consumes about 16K.

Again, those are the *minimum* requirements. You might need more. The first question to ask is: how many kilobytes (or megabytes) do you want to store on the disk?

For example, suppose you have 5000 customers, and for each customer you have 2 long typewritten lines of information (including name, address, phone number, amount due, etc.). Figuring that each "long typewritten line" holds 80 characters, you therefore have 160 characters per customer; multiply that by 5000 customers, and you get a grand total of 800,000 characters. Since a "character" is the same thing as a "byte", you have 800,000 bytes. Since a thousand bytes is called a "kilobyte", you have 800 kilobytes.

You can put the 800 kilobytes on one large disk. Or you can use two small disks, each containing 400 kilobytes. Or you can use ten teeny disks, each containing 80 kilobytes. The easiest choice is to use one big disk (800 kilobytes), and put it in a drive that can handle an 800-kilobyte disk; but such a drive is expensive. To save money, you can use a smaller drive instead, and pop many small disks in and out of the small drive; but popping small disks in and out requires manual labor which takes time, and also increases the probability of error: you might accidentally pop in the wrong disk at the wrong time. Also, if you use many small disks, you must make sure the program is intelligent enough to print messages on the screen that say "Thank you. Now please insert another disk."

Using many small disks is reasonably easy, if you're processing all the customers *sequentially* (starting with the first customer, and continuing to the last customer). But if you want to skip around— if you want to look at the first customer, then skip to the 3000th, then hop back to the 850th, etc.— you'll have to switch the small disks many times, which means you'll probably get confused and make an error. So for *sequential* access, small disks are okay; but for *random access* (which means you're skipping around), you'll need a single large disk drive instead.

Another question to ask is: how many of your customers are *active*? If half of your customers are dead accounts that you rarely have to worry about, put those customers on a separate disk and store it in a filing cabinet. In computer lingo, disks and tapes that are rarely used are called *archives*. If your entire data base requires 800K, but half of the customers are needed only for the archives, make your decision about "which disk drive to buy" as if you had only 400K to worry about.

I said that the printer should have at least 80 columns (i.e., it should handle paper wide enough to hold at least 80 characters across). If you want the computer to print large tables for financial analysis, 80 columns might not be enough: you might have to insist on 132 columns (i.e., 132 characters across).

A *word processor* is a computer that helps you compose and edit manuscripts and business correspondence. If you want the computer to be a word processor, you'll need a printer that has a daisy wheel or thimble or golf ball: dot-matrix is *not* acceptable for word processing, unless the dot-matrix is of unusually high quality.

Those are the issues you must face, if you want the computer to handle a small business. If you want the computer to handle a *large* business, you must face additional issues— which I'll discuss later.

If you're buying a computer just for pleasure and education, congratulations: you do *not* need disk drives, you do *not* need a printer, and you do *not* need lots of main memory. Instead, you can get just a tape recorder (which is much cheaper than a disk drive), you can rely on looking at just the video screen (instead of a printer), and you can get just 24K of main memory (16K of RAM and 8K of ROM). In fact, if you're buying a computer just to "learn about computers", you can get away with just 12K of main memory (4K of RAM and 8K of ROM); that amount of memory is too small to run any fancy programs but is nevertheless adequate for fascinating experiments it's adequate for a beginning class in programming.

PRICE TRENDS

The longer you wait to buy hardware, the lower the price will be; every year, computers get cheaper. A general rule of thumb is: if you wait two years, it will cost half as much. More precisely: suppose you do comparative shopping now, and buy the cheapest hardware that adequately handles your job; if you do the same comparative shopping two years later, you'll find that the cheapest adequate model then costs half as much as the cheapest model available today.

That price reduction is not some kind of "wild idea" of mine. Practically every year, since computers began in the 1940's, the total hardware price has been dropping by 30% a year. If you take a dollar, and deduct 30% of it, you get 70¢; if you then deduct 30% of the 70¢, you get 49¢. So after two years, you're spending 49¢ instead of a dollar. And there's no sign of that trend stopping.

I do *not* mean that if model X costs $1 today, it will cost 49¢ two years from now. What I *do* mean is that two years from now, there will be a new model Y (perhaps sold by a different company), which is almost as good as model X, and which costs only 49¢. Or, more likely, model Y won't quite exist, but there will be a model A that costs about 65¢ and which is *much* better than model X; and there will be a model B that costs about 35¢ and which is only *slightly* worse than model X. As for model X itself, it will probably be selling for an excessive price (about 60¢), and is definitely a silly purchase; you'd do better to buy a model A or model B. But a few idiots will still buy model X, because, during the two intervening years, lots of people bought model X, so that model X has now become famous and obtained a "good reputation", and idiots will pay for the name.

That's what usually happens to a computer company. It begins by selling a model X that's a good deal. When the industry gets excited about model X and starts praising it, people decide that they'd like to buy model X, and don't even bother to look at the competition. As a result, competitors drop their prices quickly (30% a year), whereas the manufacturer of model X drops the price of model X more slowly (a little over 20% a year). After a few years of this trend, the competitors become much better deals than model X.

In this book, I'll explain what computer prices are like today. But remember: two years from now, the world will be entirely different. (And you'll have to buy another edition of this book.)

Some parts of the computer are purely electronic: they are just a bunch of wires and circuits. Other parts of the computer are mechanical: they consists of motors and devices that move back and forth. The price of electronics drops more rapidly than the price of mechanical parts. That means: if model E contains lots of electronics, and model M contains lots of mechanical parts, the price of model E will decrease more rapidly than the price of model M; and so more and more customers will switch over to model E.

KINDS OF PRICES

A company that makes computers is called an *original equipment manufacturer (OEM)*. To make the computers, the OEM buys computer parts from other companies.

When you read an ad, you'll sometimes see a low price followed by an asterisk. At the bottom of the ad, in tiny print, you'll see the word "OEM". That means: the low price is the price that the company charges OEM's; for *you*, the price is probably higher. To get that low price, you must usually be willing to buy in quantities of 100; and the computers you'll be buying might be stripped-down models, with the understanding that you'll be adding your own improvements, and printing your own name on the equipment.

If you want to buy just one computer, and you're not an OEM, and you want the computer to be a "complete package" (instead of stripped down), you must pay the "quantity-one end-user packaged price", which is higher than the "quantity-100 OEM mechanism price".

To get a minicomputer or maxicomputer, you can buy, rent, or lease. "Rent" means you pay a certain amount each month, for at least 1 year. "Lease" means you pay a certain amount each month, for at least 2 years. The monthly payment on a lease is less than the monthly payment on rental. When you rent or lease, the computer does *not* belong to you; it belongs to the computer company. When you rent or lease, the computer company automatically takes care of repairs; when you buy, the computer company does *not* automatically take care of repairs, unless you pay for a "service contract", which costs extra, and which you'll probably want. To decide whether to buy, rent, or lease, ask your accountant, who'll tell you about "tax-deductible business expenses", "investment tax credits", and "amortization".

To get a microcomputer, you must buy. If you don't have the cash immediately, you can sometimes buy on the "installment plan", though you'd do better by getting a loan from your bank. Most microcomputers come with a 90-day warranty: if the computer needs repairs during those 90 days, return it, and the company will promptly fix it and give it back to you. After the 90 days are over, you can buy a service contract; but since microcomputers are so cheap and so rarely break down, and since fixing them is so easy, don't waste your money on a service contract: wait until the computer breaks down, and then return it to the company you bought it from and pay the one-time repair charge.

Some "discount houses" buy computers (and parts of computers) in large quantities (and get large discounts), and then resell them at 10%, 15%, or 20% below the list price. If you buy there instead of directly from the manufacturer, you'll save money.

Remember that whoever is selling you the computer is trying very hard to "make a sale". If you can prove to him that another company is selling a computer of equal quality at a slightly lower price, and that you're thinking of buying from that other company, he'll probably lower his price, just for you. Then go to the other company, tell the same story, and *that* company will lower *its* price, to match. By going back and forth between the companies several times, you can sometimes knock a huge chunk off the price. For example, one of my students managed to knock off 40%; so as a result of a few phone calls, she saved many thousands of dollars. Life gets fun when you pit the manufacturer against a discount house that sells exactly the same equipment. This whittle-'em-down approach works mainly with minicomputers and maxicomputers; microcomputer dealers work on slimmer profit margins, and therefore aren't able to lower their prices as much.

When a huge organization (such as a university or a government) wants to buy a computer, the organization sends out a *request for proposal (RFP)*, which is a memo explaining what computer capabilities the organization needs. The organization sends the memo to each of the major computer companies, and then waits to see which companies come in with the lowest bids. The lowest bidders are compared on the basis of price *and quality*. If a second round of bidding is allowed, it yields prices that are even lower.

WORD PROCESSING

WHAT IS WORD PROCESSING?

A *word-processing program* helps you write memos or reports or books; it also helps you edit what you wrote. It can help you write *anything* to *anybody* about *any* topic.

HOW A HIGH-QUALITY WORD PROCESSOR WORKS

The memo or report or book that you're writing is called the *document*. Here's how to write a document, by using a high-quality word-processing program.

The typical high-quality word-processing program comes on a disk. To begin using the word-processing program, you must put the disk into the disk drive. Then you press a few keys and type a few commands, that make the computer use the disk and start doing word processing.

You type on the computer's keyboard, as if you were typing on a typewriter. Whatever you type appears on the computer's screen. When you reach the screen's right margin, do *not* press the RETURN key; instead, just keep typing. When you try to type a word that goes past the right margin, the computer will *automatically* move the entire word down to the line below, and you'll find yourself typing on the line below. Do *not* press the RETURN key, until you reach the end of the entire paragraph.

At the end of the paragraph, press the RETURN key. That tells the computer to begin a new paragraph. If your word-processing program is of the highest quality, it will automatically indent at the beginning of the new paragraph.

While you type, you'll see a weird blip on the screen. Typically, the blip is a tiny, white, blinking rectangle. (If your computer is abnormal, the blip might be a different color, and might be an underline instead of a rectangle, and might not blink.) The blip is called the *cursor*. It appears at the spot on the screen where you're typing. Each time you type a character, the blip moves ahead, to mark the spot where the next character will appear on the screen.

On your keyboard, four of the keys have arrows on them. One of the keys has an arrow that points to the left; another key has an arrow that points to the right; another key has an arrow that points up; the fourth key has an arrow that points down. If you press one of those keys, the cursor (the blip) will move in the direction indicated by the arrow. So by using all four keys, you can move the cursor in all four directions.

If you tap one of those arrow keys quickly, the cursor will move just slightly in the direction indicated by the arrow. If you hold down an arrow keys for several seconds, the cursor will move far in the direction indicated by the arrow, and will keep moving until you lift your finger from the key.

If you make a typing mistake, use the arrow keys to move the cursor to where you made the mistake. Then retype the passage that was incorrect.

When you've finished typing and correcting your document, you give a special command that makes the computer copy the document onto a disk. Then you give another special command that makes the computer copy the document onto the printer's paper.

Each of those wonderful features has a name. The keys that have arrows on them are called *cursor keys*. If you can move the cursor far by holding down an arrow key for a few seconds (instead of having to tap the arrow key repeatedly or having to press a REPEAT key), the arrow key is said to *automatically* repeat, or *auto-repeat*. When you get near the screen's right edge and try to type a word that's too long to fit on the screen, if the computer automatically moves *the entire word* to the line below, the word-processing program is said to have *word wrap*. If the word-processing program puts your document onto a disk (instead of a tape), the word-processing program is said to be *disk-based*.

So a high-quality word-processing program is *disk-based*, has *word wrap*, and has *cursor keys* that *auto-repeat*.

SUPER SCRIPSIT

WHAT'S SUPER SCRIPSIT?

My favorite word-processing program is *Super Scripsit*. Like every other high-quality word-processing program, it's disk-based, has word wrap, and has cursor keys that auto-repeat. It also has many other nice features that make it easy to learn and easy to use. And it includes many powerful commands that let you handle the most advanced tasks.

Because Super Scripsit is so wonderful, I use it all the time. In fact, I'm using it right now, to type this chapter!

Let's see what makes Super Scripsit so great, and then compare it to its competitors.

Unfortunately, Super Scripsit runs only on Radio Shack computers. Its predecessor (which was called *Scripsit 2.0*) ran on Radio Shack's models 2, 12, and 16. The current version of Super Scripsit runs on Radio Shack's models 1, 3, and 4. Radio Shack is preparing an even fancier version, which will run on only Radio Shack's model 4.

I'll describe the current version (which works on models 1, 3, and 4).

HOW TO MOVE THE CURSOR

To move the cursor a distance of 1 character in any direction, tap any of the four arrow keys. To move the cursor a distance of *several* characters, hold down an arrow key for a few seconds: it auto-repeats.

To move the cursor *very* far in any direction, hold down an arrow key while also holding down the SHIFT key. For example, a shifted left-arrow makes the cursor hop far to the left: it hops to the left margin. A shifted right-arrow makes the cursor hop to the right, to either the right margin or the next tab stop (whichever comes first). A shifted up-arrow makes the cursor hop up to the beginning of the whole document. A shifted down-arrow makes the cursor hop down to the end of the whole document.

To move the cursor a *moderate* distance in any direction, hold down an arrow key and a letter of the alphabet simultaneously. For example, if you hold down the right-arrow while also holding down the W key, the cursor will move to the right, to the next word. W means the next word, P means the next page, V means the next video screenful (which is usually shorter than a page), G means the next 'graph (i.e., paragraph), L50 means line 50 of the document, N3 means numbered page 3 of the document, and SLOVE results in searching for the next occurrence of the word LOVE in the document. If you give those commands while holding down the right-arrow or down-arrow, the cursor moves ahead; if you give those commands while holding down the left-arrow or up-arrow, the cursor moves backwards, so you can search backwards for the previous word or page or video screenful or paragraph or even for the previous occurrence of the word LOVE!

CONTROL KEY

The keyboard includes a CONTROL key. (On the Radio Shack model 1 and model 3, the key that says "@" acts as the CONTROL key.) If you hold down the CONTROL key and a letter simultaneously, the computer will do something fancy. For example, to change the *margins*, type CONTROL-M: it makes the computer ask how you'd like to change the margins, and then changes them the way you want. To change the *tab* stops, type CONTROL-T. To *delete* characters from the middle of the document, type CONTROL-D. To *insert* characters into the middle of the document, type CONTROL-I. To *center* a title (or even a whole paragraph!), type CONTROL-C. To *print* the whole document onto paper, type CONTROL-P. To print a *form* letter to everybody on a mailing list, type CONTROL-F. There are many other CONTROL letters also, which make the computer perform feats that are even fancier! If you ever forget how to use Super Scripsit and want *help*, just type CONTROL-H: it makes the computer print a helpful seven-page summary of all of Super Scripsit's commands!

Let's look at some of those commands in more detail. . . .

HOW TO DELETE

Suppose you want to delete a character from the middle of the document. First, move the cursor to the character that you want to delete. (To move the cursor there, use the arrow keys.) Then tell the computer you want to *delete*: type a CONTROL-D. (To do that, tap the D key while holding down the CONTROL key.) Presto, the character that you wanted to delete disappears! Moreover, all the other characters on the screen move around, to fill up the space that the missing character had left; so you don't see any "hole".

To delete a whole *word* from the middle of the document, move the cursor to the beginning of the word; then tap the D key several times, while holding down the CONTROL key. Each time you tap the D key, another character disappears, until finally the whole word is gone.

To delete a whole *phrase* from the middle of the document, move the cursor to the beginning of the phrase; then hold down the D key and the CONTROL key simultaneously. The longer you hold them down, the more characters will disappear: CONTROL-D auto-repeats!

HOW TO INSERT

Of all the word-processing programs being sold, Radio Shack's Super Scripsit has the best method of letting you insert a phrase in the middle of a document. Here's how it works. To insert a phrase in the middle of the document, move the cursor to the spot where you want to insert the phrase, and then tell the computer you want to *insert*, by typing a CONTROL-I. Presto, all the other characters move out of the way, to create a huge space for you to type your phrase! Then go ahead and type your phrase. If your phrase is very long—so long that it fills the entire space—the computer will automatically move more characters out of the way, to give you an even *larger* space. So go ahead, keep typing, as you watch the computer repeatedly give you extra space. When you finish typing the phrase that you wanted to insert, you can *delete* any excess space that's left, by tapping CONTROL-D just once: it makes the computer eliminate all the extra space, and your document will look perfect!

That method of inserting has four advantages over competing systems:

1. The insertion can be as long as you wish. It can even be several pages long! (Competing systems limit your insertion to just one screenful.)

2. The computer handles the insertion very quickly. No matter how fast you type, the computer will keep up with you and update the screen immediately. (Competing systems handle insertions so slowly that you must wait for the computer to catch up with your typing.)

3. While you're inserting, the screen still shows you the part of the document that comes after the gap, so that you can see the context in which you're inserting. (In competing systems, the gap consumes the entire bottom half of the screen, and you haven't the faintest idea of what comes after the gap until you end the insertion.)

4. While you're inserting, you can use *all* of Super Scripsit's commands. You can use all the commands that involve the arrow keys; you can also use all the commands that involve the CONTROL keys. You can even use the arrow keys to hop to a totally different part of the document, do something totally unrelated to the insertion, or even start inserting *other* passages—and later come back to the original insertion and continue working on it. (Competing systems don't let you use the arrow keys while you're inserting.)

HOW TO HANDLE A BLOCK

In your document, a long passage is called a *block*. A block can be several words, or several sentences, or several paragraphs, or even several pages.

You can do 9 things to the block. . . .

1. You can *move* the block to a different part of the document.
2. You can *copy* the block, and put the copy in a different part of the document—so that the original block and the copy are both in the document.
3. You can *print* the block onto paper.
4. You can *delete* the entire block.
5. You can change the block's *linespacing*: You can make the linespacing be single-space, double-space, triple-space, 1½-space, 2½-space, or even 3½-space.
6. You can *search* through the block, to find specific words or phrases.
7. You can *hyphenate* the words in the block, to help make the right margin straight.
8. You can *freeze* the block, to prevent it from being accidentally changed.
9. You can *adjust* the block, to give it the same margins and tabs as another part of the document.

To do any of those nine activities, you must begin by telling the computer which part of the document is the "block". Here's how. . . .

First, move the cursor to the first character of the block. Then type a CONTROL-S; that tells the computer where the block *starts*.

Move the cursor to the character after the last character of the block. Then type a CONTROL-E; that tells the computer where the block *ends*.

Then type a CONTROL-B; that tells the computer you want to perform some *block action*.

The computer will ask you which of the 9 actions you want to perform. Type the first letter of the action's name. For example, if you want to *move* the block, type the letter M; if you want to *copy* the block, type the letter C; if you want to *print* the block, type the letter P; if you want to *delete* the block, type the letter D; etc.

If you choose *move* or *copy*, you must tell the computer *where* to move or copy the block. To do that, move the cursor to the place where you want the block moved or copied, and then type CONTROL-R, which makes the computer *recall* the block.

In Super Scripsit (unlike inferior word processors), the block can be many pages long, and you can even move it to a different document!

HOW TO WRITE TO DISK

When you start typing a document, the computer keeps it in the RAM. When you type a *CONTROL-W*, the computer *writes* a copy of the document onto the disk, so that the RAM and the disk *each* contain the document. If you continue to edit the document, the computer edits the version that's in the RAM. Whenever you want to copy the revised version from RAM to disk, type a CONTROL-W again.

When you've finished editing the entire document and want to *quit*, type a CONTROL Q. That makes the computer automatically do a CONTROL-W (it writes the entire document onto the disk) and also lets you start editing a different document instead.

If you type a document that's too long to fit in the RAM, the computer automatically does a CONTROL-W, to copy the document from the RAM to the disk. (The disk can hold a longer document than the RAM.) The computer leaves in the RAM the part of the document that you're editing at the moment. If you tell the computer to move to a different part of the document (by pressing the up-arrow or down-arrow keys), the computer automatically retrieves the part of the document that's on the disk, copies it back to the RAM, and puts a different part of the document onto the disk instead. This whole process is handled by the computer automatically: you don't know (and don't care) which parts of the documents are in the RAM and which parts are on the disk. In other words, the computer treats the disk as if it were extra RAM. A "disk that acts as extra RAM" is called *virtual memory*. Super Scripsit is one of the few word processors that permits virtual memory.

Whenever the computer transfers information to or from the disk, it prints this message on the screen:

```
PLEASE WAIT A MOMENT
```

That's because transferring information to or from the disk is slow: it takes several seconds. The computer refuses to let you do any typing or editing until the transfer is finished.

Fortunately, the computer will warn you whenever it's planning to do a transfer. For example, if the RAM is almost full and has only 300 free bytes left, and you nevertheless continue to type, the computer will print this warning message:

```
300 BYTES LEFT
```

That warns you the RAM has just 300 free bytes left. As you continue to type, the computer's warning message changes: the 300 decreases. When it reaches 0, the computer says PLEASE WAIT A MINUTE and does a disk transfer.

If you type a CONTROL-W before the computer reaches 0, the computer will do the disk transfer when you type the CONTROL-W, instead of when the computer reaches 0. So whenever you feel like taking a coffee break, type a CONTROL-W: that makes the computer do the disk transfer during your coffee break, instead of in the middle of your train of thought.

ADVANCED FEATURES

Super Scripsit can handle underlining, boldface, subscripts, and superscripts (for footnotes).

It can do fancy spacing. For example, it can do *proportional spacing* (so that the letter "W" is very wide and the letter "i" is very narrow). It can insert tiny extra spaces between words; each tiny extra space is just a 120th of an inch wide; that's called *microspacing*. It can make the right margin perfectly straight (by inserting extra spaces between words); that's called *right justification*. Super Scripsit is one of the few word-processing programs that can do proportional spacing, microspacing, and right justification *simultaneously*.

Super Scripsit will number each page, and will even let you create your own header to put at the top of each page, and your own footer to put at the bottom of each page. You can even create a header and footer just for the odd-numbered pages, and a different header and footer for the even-numbered pages. You can even print two columns of text side-by-side on the same page, so that the page looks like a newspaper.

Super Scripsit lets you define CONTROL-1 to stand for anything you like. For example, you can make CONTROL-1 stand for your own name and address. Then whenever you type a CONTROL-1, the computer will automatically print your name and address. If you're a lawyer, you should make CONTROL-1 stand for a popular legal phrase. You can even make CONTROL-1 stand for your own word-processing command, such as "delete the next three paragraphs". Yes, CONTROL-1 can stand for any sequence of keystrokes that you wish! Similarly, you can make CONTROL-2 stand for a second sequence of keystrokes, CONTROL-3 stand for a third sequence of keystrokes, etc.

As for myself, I make CONTROL-1 stand for a certain typesetting code, CONTROL-2 stand for a different typesetting code, etc.

Altogether, by using the CONTROL key, the CLEAR key, the SHIFT key, and the numbered keys, you can create 30 of your own codes.

Super Scripsit does all those things—plus lots more! Yet Super Scripsit costs very little: just $199.

For an additional $149, you can make it also check your spelling. The $149 gets you a disk that contains a dictionary. After you've typed your document, you give special command that makes the computer proofread your document: it checks every word of your document against the dictionary. If your document contains any word that's not in the dictionary, the computer warns you, and gives you three choices: you can correct the misspelling, or ignore the misspelling, or (if the word is a newly invented word that's not yet in the dictionary) you can insert the word into the dictionary. Radio Shack's dictionary contains 73,000 words; you can insert 2,045 additional words. You should spend the $149 to buy the dictionary, for three reasons: it will find your spelling errors; it will find your typographical errors; and it will find passages that are too awkward to be understood (because they contain unnatural words that aren't in the dictionary). Since the computer looks in its dictionary to find *every* word you typed, it's a much more thorough proofreader than the average human!

OTHER WORD-PROCESSING SYSTEMS

Here are some feats that other word-processing systems can perform:

put footnotes at the bottom of each page
make a table of contents
make an index, in alphabetical order
do mathematical calculations, and put the results into your document
type scientific formulas
show you, on the screen, two different parts of your document simultaneously
let you undo your last command, if it wrecked your document
suggest alternative words, by using a disk that contains a thesaurus
correct your style, by using a disk that lists common stylistic errors
correct your hyphenation, by using a dictionary that includes hyphenation
correct your punctuation

Yes, each of those fancy tasks *can* be done by other word-processing programs! Unfortunately, those programs are harder to learn how to use.

So if you don't need such fanciness, I recommend that you use with Super Scripsit, which is the easiest word-processing system that's reasonably powerful.

DATA MANAGEMENT

WHAT'S A DATA-MANAGEMENT SYSTEM?

A *data-management system* is a program that manipulates lists of facts. For example, it can manipulate a mailing list, a phone directory, and any other lists you wish, concerning your customers, suppliers, employees, friends, enemies, students, teachers, clients, libraries, and hobbies; it puts all that data about your business and life onto the disk, and then lets you retrieve that data easily. It turns your computer into an electronic filing cabinet.

It's called a *data-management system* or *file-management system* or *information-retrieval system*; the terms are synonymous.

FILE-CABINET JARGON

Suppose you walk into an old-fashioned office that doesn't have a computer. In that office, you'll find a filing cabinet that contains several drawers. One drawer's labeled "customers"; another drawer's labeled "employees"; another drawer's labeled "suppliers". Each drawer contains many "index cards", alphabetized.

For example, the drawer labeled "customers" contains a card about each customer; the first card might be labeled "ADAMS, JOAN"; the last card might be labeled "ZABRONSKI, JILL". The first card contains all known information about Joan Adams: it contains her name, address, phone number, a list of everything she's purchased from the office, how much she paid, how much she still owes, and other personal information about her. That card is called her *record*. Each item of information on that card is called a *field*. If the card is a pre-printed form, it allows a certain amount of space for each item: for example, it might allow only 30 characters for the person's name. The number of characters allowed for a field is called the *field's width*. In that example, the width of the NAME field is 30 characters.

Each drawer is called a *file*. For example, the drawer that contains information about customers is called the *customer file*; another drawer is the *employee file*; another drawer is the *supplier file*.

The entire filing cabinet—which contains all the information about your company—is called the *data base*.

A SAMPLE FILE

Here's a file, about amazing students in the School of Life:

Last name: Smith *First name: Suzy*
Age: 4 *Class: 12*
Comments: Even though she's only 4 years old, she's already in the twelfth grade and will graduate this year, because she's a fast girl—she really knows how to move.

Last name: Kosinski *First name: Stanislaw*
Age: 21 *Class: 10*
Comments: He never quite made it. He's a bit slow, afraid of Polish jokes, and has a crush on the teacher.

Last name: Ketchopf *First name: Heinz*
Age: 57 *Class: 1*
Comments: His pour grades make him the slowest Ketchopf in the West.

Last name: Nixon *First name: Tricky Dick*
Age: 70 *Class: 13*
Comments: The unlucky President, he disappointed our country.

Last name: Walter *First name: Russy-poo*
Age: 36 *Class: 0*
Comments: This guy has no class.

Last name: Smith *First name: Buffalo Bob*
Age: 7 *Class: 2*
Comments: Boringly normal, he's jealous of his sister Suzy.

That file consists of six records: Suzy Smith's record, Stanislaw Kosinski's record, Heinz Ketchopf's record, Tricky Dick Nixon's record, Russy-poo Walter's record, and Buffalo Bob Smith's record. Each record consists of five fields: last name, first name, age, class, and comments. The age and class fields are narrow; the comments field is wider.

DATA MANAGEMENT VERSUS WORD PROCESSING

A word-processing system lets you write a memo, put the memo onto a disk, edit the memo, and copy the memo onto a printer. A data-management system is quite similar. Like a word-processing system, a data-management system lets you write information, put the information onto a disk, edit the information, and copy the information onto a printer.

In a word-processing system, the information is called a *document*, which consists of paragraphs, which in turn consist of sentences. In a *data-management* system, the information is called a *file* (instead of a document): the file consists of records, which in turn consist of fields.

Because a data-management system is so similar to a word-processing system, a word-processing system can act as a crummy data-management system. But a *good* data-management system offers the following "extras", which the typical word-processing system lacks. . . .

A good data-management system can alphabetize.

A good data-management system can automatically rearrange information, to put it in numerical order.

A good data-management system can check for criteria. For example, you can tell a good data-management system to check which customers are women under 18 who have light red hair and live in a red-light district. You can even make the data-management system print their names and addresses on mailing labels, in zip code order. You can even make the data-management system create your own phone book, containing the names and phone numbers of just those women. As you can see, data-management systems are very potent, and are excellent tools for invading your victims' privacy!

PFS

WHAT'S PFS?

The nicest data-management system is called *Personal Filing System (PFS)*: it's a program that you can buy for Apple, Radio Shack, and IBM computers. It comes on disk. It's popular, because it's easy to use and also very flexible.

For several years, PFS has been the best-selling data-management system for Apple computers. I expect the newer versions, for Radio Shack and IBM, will become popular also.

The fanciest version is the one for the IBM PC. The following explanation applies only to that version (though the Apple and Radio Shack versions are somewhat similar).

FOUR DISKS

The IBM PC version of PFS comes on four disks.

The main disk is called *PFS File*: it performs the most important data-management functions. The second disk is called *PFS Report*: it prints tables and does mathematical calculations (such as "find the average"). The third disk is called *PFS Graph*: it draws pretty colorful graphs that summarize all the data in the file. The fourth disk is called *PFS Write*: it's a word-processing program that writes memos to and about the people in the file.

Each disk lists for $140, except for PFS Report (which lists for only only $125). You can pay $95 instead of

Each disk lists for $140, except for PFS Report (which lists for only $125). You can pay $95 instead of $140, and pay $84 instead of $125, by contacting a discount dealer such as Conroy-La Point (Box 23068, Portland, OR 97223, 503-245-6200 or 800-547-1289).

If you don't have enough money to buy all four disks, buy just the main disk (PFS File): it includes all the essentials. Here's how it works. . . .

HOW TO GET STARTED

When you start using PFS File, this list appears on the screen:

```
1. Design file
2. Add
3. Copy
4. Search/update
5. Print
6. Remove
7. Exit PFS File
```

That list shows the 7 tasks that PFS File can perform. To make the computer do one of those tasks, type a number from 1 to 7. That list of tasks is called a *menu*, because it's like a Chinese restaurant where you order by number.

If this is your first time using PFS File, you must design a new file, so type the number 1. (You must also hit a few extra keys, which are explained in the booklet that comes with PFS File.)

After typing the number 1 (and hitting a few extra keys), most of the screen will become blank. On the blank screen, type a form, which you'll fill in later. For example, if you want to store each student's last name, first name, age, and class, plus comments, type this form:

```
LAST NAME:                              FIRST NAME:
AGE:              CLASS:
COMMENTS:
```

Notice you should type the word LAST, then a space, then the word NAME, then a colon, then several spaces, then the word FIRST, then the word NAME, then a colon, then press the RETURN key, etc. That form creates five fields: last name, first name, age, class, and comments. When you've finished typing the form, press the F10 key.

The computer will print the menu again. So again you must choose a number from 1 to 7. This time, try choosing 2 (which lets you add specific students to the file).

When you choose 2, the computer will automatically show your form again on the screen, like this:

```
LAST NAME:                              FIRST NAME:
AGE:                CLASS:
COMMENTS:
```

Then fill in the blanks. For example, if one of the students is Suzy Smith, fill in the blanks like this:

```
LAST NAME: Smith                        FIRST NAME: Suzy
AGE: 4              CLASS: 12
COMMENTS: Even though she's only 4 years old, she's already in the twelfth grade
and will graduate this year, because she's a fast girl--she really knows how
to move.
```

To fill in the blanks that way, type Smith, then press the TAB key (which moves the cursor to the next field), then type Suzy, then press the TAB key, then type 4, then press the TAB key, then type 12, then press the TAB key, then type the commentary sentence.

When you've finished filling in Suzy Smith's form, press the F10 key. The computer will show the blank form again:

```
LAST NAME:                              FIRST NAME:
AGE:                CLASS:
COMMENTS:
```

Fill in the blanks again, for the next student (Stanislaw Kosinski). Then press the F10 key again. Repeat that process for each student. As you type, the computer puts all that information onto a disk. When you've finished the last student, press the F10 key again; then press the ESCAPE key. The entire file is now on the disk, and you're back at the menu. You must choose a number from 1 to 7 again.

HOW TO COPY

If you choose 3, the computer will copy the data to a second disk. The second disk is a backup, to be used only if the original disk gets damaged. After copying, the computer returns you to the menu again, and you must choose a number from 1 to 7 again.

If you choose 4, the screen will show the blank form again, like this:

```
LAST NAME:                          FIRST NAME:
AGE:              CLASS:
COMMENTS:
```

If you fill in a few of the blanks, the computer will fill in the rest. For example, if you fill in just the LAST NAME (Smith) and then press the F10 key, the computer will find everybody whose last name is Smith, and will show you each Smith's record, one at a time.

Whenever the computer finds a Smith's record, it pauses, to let you look at the record. When you've finished looking, press the F10 key, which makes the computer find the next Smith.

While you're looking at a Smith's record, you can revise the record. (Revising is also called *editing* or *updating*.) To revise the record you're looking at, move the cursor to the field you want to revise, by tapping the TAB key several times. Then retype that field.

If you tell the computer to search for "S.." instead of "Smith" (by putting "S.." in the LAST NAME field), the computer will find everybody whose last name begins with "S". If you say "..th", the computer will find everybody whose last name ends with "th". If you say "..m..", the computer will find everybody whose name contains an "m". If you say "/Smith", the computer will find everybody whose name is *not* Smith. If you say "/..m..", the computer will find everybody whose name does *not* contain an m.

Suppose you want to find Kosinski's record, but you can't remember how he spells his name. You don't remember whether it's "Kosinski" or "Cosinski" or "Kozinski" or "Kosinscki" or "Kosinsky"; you remember merely that it includes the letters "in". If you put "..in.." into the LAST NAME field, the computer will find everybody whose name contains the word "in"—and it will find Kosinski's record.

If you fill in just the AGE, the computer will find everybody that has that AGE. To find everybody who's 10 years old, put "=10" in the AGE field. To find everybody who's less than 10, put "<10" in the AGE field. To find everybody who's greater than 10, put ">10". To find everybody who's between 10 and 18, put "=10..18". To find everybody who's *not* between 10 and 18, put "/=10..18". To find everybody whose age was left blank, put "/..".

If you fill in just the COMMENT, the computer will find everybody that has that COMMENT. For example, if you put "..slow.." into the COMMENT field, the computer will find everybody who's COMMENT mentions "slow": it will find Stanislaw Kosinski (who's "a bit slow") and will also find Heinz Ketchopf (who's "the slowest Ketchopf in the West").

Suppose you want to find everybody who's old and stupid. Specifically, suppose you want to find everybody whose age is over 20 and who still hasn't graduated from the eighth grade. In the AGE field put ">20", and in the CLASS field put "<9". The computer will find Heinz Ketchopf (who's 57 years old and in the first grade) and Russy-poo Walter (who's 36 years old and whose CLASS is 0).

When the computer has finished finding the records that you asked for, it will return you to the menu, and you must choose a number from 1 to 7 again.

ADVANCED FEATURES

To print on paper, choose option 5 from the menu. To delete the records of students who died, choose option 6 from the menu. Whenever you've finished using PFS File, choose option 7.

PFS File is extremely flexible. For example, it can print on regular paper or on mailing labels. When it prints, it can print the records in alphabetical order or in ZIP code order or in any other order you like. It can hide any information you wish: for example, you can make it print a person's address but hide the phone number, so that the phone number doesn't appear on paper. The file can be up to 4 megabytes long, and can have up to 32,000 records. Each record can have as many fields as you wish, and each field can be as long as you wish. A record can be even longer that the computer's screen: the record can include several screenfuls of information. Each screenful is called a *page*; the record can include several pages. If one of your customers is so weird that you want to store extra information about him, you can add an *attachment* to his record: the attachment can be several pages long. If you've created a whole file and later change your mind and decide that you'd like to insert extra fields in the middle of each record, you can do so.

PFS File is *efficient*: it works quickly, and uses very little space on your disk.

LIMITATIONS

I like PFS because it's easy to learn how to use, flexible, and inexpensive. But PFS isn't perfect: it lacks some advanced features. Here's what PFS lacks. . . .

Although PFS lets you update a customer's record, it can't do math as part of the updating. For example, suppose a customer's record includes the "balance due", and the customer mails you a check that reduces the amount of the "balance due". PFS can't perform the subtraction. Instead, you must buy a pocket calculator, subtract the customer's check from the balance due, and then feed the calculator's answer to PFS.

PFS can't check whether your input is reasonable. PFS won't gripe if you tell it that a person's age is 172, or that a person's weight is 985 pounds, or that a person sent a check for $.007, or that a person's phone number is 26-8128, or that a person's sex is N (instead of M), or that a person's middle initial is 7 (instead of a letter). More important, it won't gripe if you tell it to print a paycheck for $1,000,000.

If your file is too big to fit on a disk—if your file requires *two* disks—PFS can't handle it.

PFS can't handle relationships between different files. For example, suppose two different files contain information about Joan Adams. PFS can't merge those files together, to create a single file. Moreover, if Joan Adams is an old-fashioned woman who gets married and changes her name, PFS can't automatically change her name in both files; you must tell PFS about her name change *twice*. A program that *can* handle relationships between different files is called a *data base management system (DBMS)*; PFS is not a DBMS. PFS is a *data-management system* but not a data-*base* management system.

If the limitations of PFS bother you, buy a data-management system that's fancier. Unfortunately, the fancier ones are more expensive and are also harder to learn how to use.

Examples of fancier data-management systems are *DB Master* (which is somewhat fancy) and *Dbase 2* (which is even fancier). DB Master works on the IBM PC and the Apple; Dbase 2 works on the IBM PC and on CP/M computers.

CHAPTER 3

ADVANCED PROGRAMMING USING BASIC

STRANGE THINGS

EXTRACTING FROM STRINGS

Let's analyze the word "SMART".

Since "SMART" has 5 characters in it, the *length* of "SMART" is 5. If you say—

```
PRINT LEN("SMART")
```

the computer will print the LENgth of "SMART"; it will print:

```
 5
```

The left two characters of "SMART" are "SM". If you say—

```
PRINT LEFT$("SMART",2)
```

the computer will print:

```
SM
```

Try this program:

```
10 A$="SMART"
20 PRINT LEFT$(A$,2)
```

Line 10 says A$ is "SMART". Line 20 says to print the left 2 characters of A$, which are "SM". The computer will print:

```
SM
```

If A$ is "SMART", here are the consequences. . . .

LEN(A$) is the LENgth of A$. It is 5.
LEFT$(A$,2) is the LEFT 2 characters of A$. It is "SM".
RIGHT$(A$,2) is the RIGHT 2 characters of A$. It is "RT".
MID$(A$,2) is the MIDdle of A$, beginning at the 2nd character. It is "MART".
MID$(A$,2,3) begins at the 2nd character and includes 3 characters. It's "MAR".

CHANGING THE MIDDLE

You can change the middle of a string, like this:

```
10 A$="BUNKERS"
20 MID$(A$,2)="OWL"
30 PRINT A$
```

Line 10 says A$ is "BUNKERS". Line 20 changes the middle of A$ to "OWL"; the change begins at the 2nd character of A$. Line 30 prints:

```
BOWLERS
```

Here's a variation:

```
10 A$="BUNKERS"
20 MID$(A$,2,1)="OWL"
30 PRINT A$
```

Line 10 says A$ is "BUNKERS". Line 20 changes the middle of A$; the change begins at the 2nd character of A$; the ",1" makes the computer use only 1 letter from "OWL". Line 30 prints:

```
BONKERS
```

Another variation:

```
10 A$="BUNKERS"
20 MID$(A$,2)="AD AGENCY"
30 PRINT A$
```

Line 10 says A$ is "BUNKERS". Line 20 says to change the middle of A$, beginning at the 2nd character of A$. But "AD AGENCY" is too long to become part of "BUNKERS". The computer uses as much of "AD AGENCY" as will fit in "BUNKERS"; the computer will print:

```
BAD AGE
```

ADDING STRINGS

You can add strings together, to form a longer string:

```
10 A$="FAT"+"HER"
20 PRINT A$
```

Line 10 says A$ is "FATHER". Line 20 makes the computer print:

```
FATHER
```

SEARCHING IN A STRING

You can make the computer search in a string, to find another string. To make the computer search IN the STRing "NEEDED", to find "ED", say:

```
PRINT INSTR("NEEDED","ED")
```

Exception: Apple & Commodore don't understand INSTR.

Since "ED" begins at the third character of "NEEDED", the computer will print:

```
 3
```

If you say—

```
PRINT INSTR("NEEDED","EY")
```

the computer will search in the string "NEEDED" for "EY". Since "EY" is *not* in "NEEDED", the computer will print:

```
 0
```

If you say—

```
PRINT INSTR(4,"NEEDED","ED")
```

the computer will hunt in the string "NEEDED" for "ED"; but the hunt will begin at the 4th character of "NEEDED". The computer finds the "ED" that begins at the 5th character of "NEEDED"; the computer will print:

```
 5
```

CLOCK

The typical computer has a built-in clock.

To set the clock to 7 seconds after 1:45PM, type this:

```
TIME$="13:45:07"
```

To set the date to January 27, 1984, type this:

```
DATE$="01-27-1984"
```

Afterwards, whenever you want to find out the current time and date, type this:

```
PRINT TIME$
PRINT DATE$
```

If you say—

```
PRINT TIMER
```

Exception: TIME$, DATE$, and TIMER work that way only on the IBM PC.

the computer will tell you how many seconds have elapsed since midnight.

When you turn the computer off, it forgets the time and date. When you turn the computer on again, tell it the new time and date.

STRING-NUMBER CONVERSION

This program converts a string to a number:

```
10 A$="72.6"
20 B=VAL(A$)
30 PRINT B+1
```

Line 10 says A$ is the string "72.6". Line 20 says B is the numeric VALue of A$; so B is the number 72.6. Line 30 prints:

```
 73.6
```

VAL converts a string to a number. The opposite of VAL is STR$: it converts a number to a string. For example, STR$(-7.2) is the string "-7.2".

STR$(81.4) is the string " 81.4". Notice that in the string " 81.4", the 8 comes after a space (instead of coming after a minus sign).

Exception: Apple omits the blank space.

ASCII

Each character has a code number. For example, the code number for "A" is 65; the code number for "B" is 66; the code number for "C" is 67; etc.

Those code numbers form the *American Standard Code for Information Interchange*, which is abbreviated *ASCII*, which is pronounced "ass key". Programmers say, "the ASCII code number for A is 65".

If you say—

```
PRINT ASC("A")
```

the computer will print the ASCII code number for "A". It will print:

```
 65
```

Similarly, if you say PRINT ASC("B"), the computer will print 66.

If you say—

```
PRINT CHR$(65)
```

the computer will print the CHaRacter whose code number is 65. It will print:

```
A
```

The code number for a capital "A" is 65, capital "B" is 66, capital "C" is 67, etc. The code number for a small "a" is 97, small "b" is 98, small "c" is 99, etc. The code number for the digit "0" is 48, the digit "1" is 49, the digit "2" is 50, etc.

Here's the complete list of code numbers, from 33 to 126:

```
Character: ! " # $ % & ' ( ) * + , - . / 0 1... 9 : ; < = >
Code #:   33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49...57 58 59 60 61 62

Character: ? @ A B... Z [ \ ] ^ _ ` a b... z { | } ~
Code #:   63 64 65 66...90 91 92 93 94 95 96 97 98...122 123 124 125 126
```

Exception: on Apple, Commodore, & Radio Shack Color Computers, codes 91-126 differ.

32 is the code number for a blank space. 127 is the code number for DELETE.

Notice that the code number for a quotation mark is 34. This makes the computer print a quotation mark:

```
PRINT CHR$(34)
```

Suppose you want the computer to print:

```
SCHOLARS THINK "HAMLET" IS A GREAT PLAY
```

To make the computer print the quotation marks around "HAMLET", use CHR$(34), like this:

```
PRINT "SCHOLARS THINK ";CHR$(34);"HAMLET";CHR$(34);" IS A GREAT PLAY"
```

On the typical computer's keyboard, you'll see a key marked CONTROL (or CTRL). That CONTROL key is next to the left SHIFT key. While you hold down the CONTROL key, you can tap the A key; that's called a *controlled A*. The code number for a controlled A is 1. Similarly, the code number for a controlled B is 2; a controlled C is 3; etc.

Holding down the CONTROL key subtracts 64 from the code number of the other key. For example, the A key is normally 65, so a controlled A is 65 64, which is 1. The B key is normally 66, so a controlled B is 66-64, which is 2.

You've seen that a controlled A has code 1, and a controlled B has code 2. Controlled @ has code 0. In fact, by using the CONTROL key, you can generate all the characters that have codes from 0 to 31.

The code numbers from 128 up to 255 are used for graphics. To explore your computer's ability to print both text and graphics, run this program:

```
10 FOR I = 33 TO 255
20     PRINT I;CHR$(I);"   ";
30 NEXT
```

The computer will print each number and its associated character; it will print:

```
 33 !    34 "    35 #    36 $   etc.
```

When the computer finally reaches 128, it will start printing graphics characters.

In that program, you can try starting I at a number lower than 33. For example, you can try starting I at 5. But be careful: some of those low numbers will make your computer act very strangely!

On the Apple, do *not* say PRINT CHR$(4). If you make the mistake of saying CHR$(4), the computer might turn on the disk drive and start wrecking your disk!

If your computer is typical, you can make it beep by saying:

```
PRINT CHR$(7)
```

If you say—

```
PRINT "LOVE"
```

the computer will print LOVE on your screen. If you say—

```
LPRINT "LOVE"
```

Exception: on Apple, Commodore, & Radio Shack Color Computers, replace LPRINT by other words.

the typical computer will print LOVE on paper (instead of your screen), by using a printer. The most popular printers are manufactured by Epson. To make the typical Epson printer do weird things on paper, give one of these commands:

Command	Meaning
`LPRINT CHR$(7);`	beep for about .3 seconds
`LPRINT CHR$(9);`	move to the right, to the next tab stop
`LPRINT CHR$(12);`	go to the top of the next page
`LPRINT CHR$(14);`	make characters very wide, 5 per inch
`LPRINT CHR$(15);`	make characters narrow, 16½ per inch
`LPRINT CHR$(15)CHR$(14);`	make characters wide, 8¼ per inch
`LPRINT CHR$(18);`	cancel CHR$(15); return to 10 per inch
`LPRINT CHR$(20);`	cancel CHR$(14)
`LPRINT CHR$(27)"0";`	8 lines per inch (instead of 6 per inch)
`LPRINT CHR$(27)"1";`	about 10 lines per inch
`LPRINT CHR$(27)"2";`	return to normal 6 lines per inch
`LPRINT CHR$(27)"3"CHR$(n);`	n/216 lines per inch
`LPRINT CHR$(27)"4";`	switch to italics
`LPRINT CHR$(27)"5";`	cancel italics; switch back to Roman
`LPRINT CHR$(27)"<";`	move all the way left, to the left margin
`LPRINT CHR$(27)"@";`	cancel all previous CHR$ commands
`LPRINT CHR$(27)"A"CHR$(n);`	n/72 lines per inch
`LPRINT CHR$(27)"C"CHR$(n);`	set page length to n lines
`LPRINT CHR$(27)"E";`	horizontal fill, so that "....." becomes "—"
`LPRINT CHR$(27)"F";`	cancel the horizontal fill
`LPRINT CHR$(27)"G";`	vertical fill, so that "¦" becomes "|"
`LPRINT CHR$(27)"H";`	cancel the vertical fill

REPEATING CHARACTERS

Suppose you want to print "BBBBBBBBBBBBBBBBBBB". Here's a short-cut:

```
PRINT STRING$(20,"B")
```

Exception: Apple & Commodore don't understand STRING$.

That tells the computer to print a string of 20 B's.

Here's a different way to accomplish the same goal:

```
PRINT STRING$(20,66)
```

That tells the computer to print, 20 times, the character whose ASCII code number is 66.

STRING$ can make the computer repeat a single character, but not a whole word. So although STRING$(20,"B") makes the computer repeat the character "B", STRING$(20,"BLOW") will *not* make the computer repeat the word "BLOW"; instead, the computer will repeat just the first character of "BLOW" (which is "B").

PRINT USING

FUNDAMENTALS

Suppose you want to add $12.47 to $1.03. The correct answer is $13.50. This almost works:

```
PRINT 12.47+1.03
```

The computer will print:

```
 13.5
```

But instead of 13.5, we should try to make the computer print 13.50.

This program forces the computer to print 13.50:

```
PRINT USING "##.##"; 12.47+1.03
```

The "##.##" is called the *picture* or *image* or *format*: it says to print two characters, followed by a decimal point, followed by two digits. The computer will print:

```
13.50
```

Exception: Apple & Commodore don't understand PRINT USING. On DEC mini & maxi, replace the semicolon by a comma.

The computer will round:

```
PRINT USING "##.##"; 300/7
```

When the computer divides 300 by 7, it gets 42.8571, but the format rounds the answer to 42.86. The computer will print:

```
42.86
```

If you say—

```
PRINT USING "FUNNY##.##JOE"; 300/7
```

the computer will print:

```
FUNNY42.86JOE
```

MULTIPLE NUMBERS

If you say—

```
PRINT USING "MY ## ENEMIES DRANK ####.# GALLONS OF BOOZE"; 24,1784.5
```

the computer will print:

```
MY 25 ENEMIES DRANK 1784.5 GALLONS OF BOOZE
```

A format, such as "###.##", is a string. You can replace it by a string variable:

```
10 A$="###.##"
20 PRINT USING A$; 247.91
30 PRINT USING A$; 823
40 PRINT USING A$; 7
50 PRINT USING A$; -5
60 PRINT USING A$; -80.3
```

The computer will print:

```
247.91
823.00
  7.00
 -5.00
-80.30
```

The computer prints the numbers so that the decimal points are underneath each other.

To print those numbers *across* instead of down, say this:

```
PRINT USING "###.##"; 247.91,823,7,-5,-80.3
```

The computer will print:

```
247.91823.00  7.00 -5.00-80.30
```

Because the computer prints those numbers so close together, they're hard to read. To make the computer insert extra space between the numbers, widen the format, by putting a fourth "#" before the decimal point:

```
PRINT USING "####.##"; 247.91,823,7,-5,-80.3
```

Now the computer will print:

```
 247.91 823.00   7.00  -5.00 -80.30
```

LARGE NUMBERS

Suppose you say:

```
PRINT USING "###.##"; 16238.7
```

The computer tries to print 16238.7, by using the format "###.##". But since the format allows only three digits before the decimal point, the format isn't large enough to fit 16238.7. So the computer must disobey the format. But the computer also prints a percent sign, which means "Warning! I am disobeying you!" Altogether, the computer prints:

```
%16238.70
```

FINAL SEMICOLON

At the end of the PRINT USING statement, you can put a semicolon:

```
10 PRINT USING "##.##"; 13.5;
20 PRINT "ZAP"
```

Line 10 makes the computer print 13.50. Because of the semicolon at the end of line 10, the computer will print ZAP on the same line, like this:

```
13.50ZAP
```

If you say—

```
10 A$="SKY######.##ART"
20 PRINT USING A$; -27931.6
30 PRINT USING A$; 8.95
```

the computer will print:

```
SKY-27931.60ART
SKY     8.95ART
```

If you change the format to "SKY###,###.##ART", the computer will insert a comma if the number is large:

```
SKY-27,931.60ART
SKY      8.95ART
```

If you change the format to "SKY######.##-ART", the computer will print the minus sign at the right instead of at the left:

```
SKY 27931.60-ART
SKY     8.95 ART
```

That minus sign, at the right, is called a *trailing minus*, because it trails after the number. A minus sign at the left is called a *leading minus*.

If you change the format to "SKY+#####.##ART", the computer will print a plus sign in front of any positive number:

```
SKY-27931.60ART
SKY    +8.95ART
```

Normally, a format begins with ##. If you begin with $$ instead (like this: "SKY$$#####.##ART"), the computer will print a dollar sign before the digits:

```
SKY-$27931.60ART
SKY     $8.95ART
```

If you begin with ** (like this: "SKY**#####.##ART"), the computer will print asterisks before the number:

```
SKY*-27931.60ART
SKY******8.95ART
```

If you begin with **$ (like this: "SKY**$#####.##ART"), the computer will print asterisks and a dollar sign:

```
SKY*-$27931.60ART
SKY******$8.85ART
```

When you're printing a paycheck, use the asterisks, to prevent the employee from enlarging his salary. Because the asterisks protect the check from being altered, the asterisks are called *check protection*.

You can combine several techniques in a single format. For example, you can combine the comma, the trailing minus, and the **$ (like this: "SKY**$##,###.##-ART"), so that the computer will print:

```
SKY**$27,931.60-ART
SKY*******$8.95 ART
```

If you change the format to "SKY##.#####↑↑↑↑ART", the computer will print numbers by using E notation:

```
SKY-2.79316E+04ART
SKY 8.95000E+00ART
```

SWAPPING

HOW TO SWAP

Modern computers understand the word SWAP:

```
10 A=4: B=9
20 SWAP A,B
30 PRINT A;B
```

Exception: SWAP works only on IBM PC, CP/M, & Radio Shack model 4.

Line 10 says A is 4 and B is 9. Line 20 swaps A with B, so that A becomes 9, and B becomes 4. Line 30 prints:

```
 9  4
```

If your computer is old-fashioned and doesn't understand the word SWAP, replace line 20 by this:

```
20 S=A: A=B: B=S
```

That example swapped numbers. You can also swap strings:

```
10 A$="HORSE": B$="CART"
20 SWAP A$,B$
30 PRINT A$;" ";B$
```

Line 10 says A$ is "HORSE" and B$ is "CART". Line 20 swaps A$ with B$, so that A$ becomes "CART", and B$ becomes "HORSE". Line 30 puts the CART before the HORSE:

```
CART HORSE
```

If your computer doesn't understand SWAP, replace line 20 by this:

```
20 S$=A$: A$=B$: B$=S$
```

Here are some cards:

Queen of Hearts
Jack of Diamonds
Ace of Spades
Joker
King of Clubs

Let's shuffle them, to put them into a random order.
To begin, put the list of cards into a DATA statement:

```
10 DATA QUEEN OF HEARTS,JACK OF DIAMONDS,ACE OF SPADES,JOKER,KING OF CLUBS
```

We have 5 cards:

```
20 N=5
```

Let the Queen of Hearts be called X$(1), the Jack of Diamonds be called X$(2), the Ace of Spades be called X$(3), the Joker be called X$(4), and the King of Clubs be called X$(5):

```
30 DIM X$(N)
40 FOR I = 1 TO N: READ X$(I): NEXT
```

Shuffle the cards, by using the following strategy. . . . Swap the card N with a random card before it (or with itself); then swap card N-1 with a random card before it (or with itself); then swap card N-2 with a random card before it (or with itself); etc. Keep doing that, until you finally reach card 2, which you swap with a random card before it (or with itself). Here's the code:

```
50 RANDOMIZE
60 FOR I = N TO 2 STEP -1: SWAP X$(I),X$(RND(I)): NEXT
```

If your computer doesn't understand SWAP, replace "SWAP X$(I),X$(RND(I))" by this:

```
R=RND(I): S$=X$(I): X$(I)=X$(R): X$(R)=S$
```

If your computer doesn't understand the word RANDOMIZE, or doesn't understand that RND(I) is a random number from 1 to I, you must make further changes.
Finally, print the shuffled deck:

```
70 FOR I = 1 TO N: PRINT X$(I): NEXT
```

The computer will print something like this:

```
KING OF CLUBS
JOKER
JACK OF DIAMONDS
QUEEN OF HEARTS
ACE OF SPADES
```

To shuffle a larger deck, change just lines 10 and 20.

ALPHABETIZING

Here are a dozen names: Sue, Ann, Joe, Alice, Ted, Jill, Fred, Al, Sam, Pat, Sally, Mike. Let's make the computer alphabetize them.

To begin, put the list of names into a DATA statement:

```
10 DATA SUE,ANN,JOE,ALICE,TED,JILL,FRED,AL,SAM,PAT,SALLY,MIKE
```

We have 12 names:

```
20 N=12
```

Let Ann be called X$(1), Joe be called X$(2), Alice be called X$(3), etc. Here's how:

```
30 DIM X$(N)
40 FOR I = 1 TO N: READ X$(I): NEXT
```

Alphabetize the names, by using the following strategy. . . . Compare the first name against the second; if they're not in alphabetical order, swap them. Compare the second name against the third; if they're not in alphabetical order, swap them. Compare the third name against the fourth; if they're not in alphabetical order, swap them. Continue that process, until you finally compare the last two names. But each time you swap, you must start the whole process over again, to make sure the preceding names are still in alphabetical order. Here's the code:

```
50 FOR I = 1 TO N-1
60     IF X$(I)>X$(I+1) THEN SWAP X$(I),X$(I+1): GO TO 50
70 NEXT
```

Finally, print the alphabetized list:

```
80 FOR I = 1 TO N: PRINT X$(I): NEXT
```

The computer will print:

```
AL
ALICE
ANN
FRED
JILL
JOE
MIKE
PAT
SALLY
SAM
SUE
TED
```

Putting words in alphabetical order—or putting numbers in numerical order—is called *sorting*. In the program above, the sorting occurs in lines 50-70. Those three short lines are called the *Short Three-Line Sort*.

In the Short Three-Line Sort, lines 50-70 form a loop. The part of the loop that the computer encounters the most often is the phrase "IF X$(I)>X$(I+1)". In fact, if you tell the computer to alphabetize a list of names that's long (several *hundred* names), the computer spends the *majority* of its time repeatedly handling "IF X$(I)>X$(I+1)".

How long will the computer require, to handle a long list of names? The length of time depends mainly on how often the computer encounters the phrase "IF X$(I)>X$(I+1)". If the list contains N names, the number of times that the computer encounters the phrase "IF X$(I)>X$(I+1)" is approximately $N^3/8$. Here are some examples:

N (number of names)	$N^3/8$ (approximate number of encounters)
10	125
12	216
20	1,000
40	8,000
100	125,000
1,000	125,000,000
10,000	125,000,000,000

For example, the chart says that a list of 12 names requires approximately 216 encounters. The 216 is just an approximation: the exact number of encounters depends on which list of names you're trying to alphabetize. If the list is nearly in alphabetical order *already*, the number of encounters will be less than 216; if the list is in *reverse* alphabetical order, the number of encounters will be more than 216; but if the list is typical (not yet in any particular order), the number of encounters will be about 216. For the list that we tried (Sue, Ann, Joe, Alice, Ted, Jill, Fred, Al, Sam, Pat, Sally, Mike), the exact number of encounters happens to be 189, which is close to 216.

How much time will your computer require, to finish sorting? The amount of time depends on which brand of computer you have, how many names are in the list, and how long each name is. A typical microcomputer (handling a list of typical names) requires .01 seconds per encounter. Multiplying the number of encounters by .01 seconds, you get:

Number of names	Encounters	Time
10	125	1.25 seconds
12	216	2.16 seconds
20	1,000	10 seconds
40	8,000	80 seconds = 1.3 minutes
100	125,000	1,250 seconds = 20.8 minutes
1,000	125,000,000	1,250,000 seconds = 14.4 days
10,000	125,000,000,000	1,250,000,000 seconds = 39.6 years

Moral: never let a 70-year-old programmer alphabetize a list of 10,000 names by using the Short Three-Line Sort—because when the computer finishes running the program (39.6 years later), the programmer will probably be dead!

In the Short Three-Line Sort, the end of line 60 says "GO TO 50". For an interesting experiment, replace the "GO TO 50" by "IF I>1 THEN I=I-1: GO TO 60", so that line 60 looks like this:

```
60     IF X$(I)>X$(I+1) THEN SWAP X$(I),X$(I+1): IF I>1 THEN I=I-1: GO TO 60
```

Even though that new, fancy version of line 60 is longer to type than the original, the computer handles it more quickly, because it tells the computer to GO TO line 60 instead of forcing the computer to go all the way back to line 50. The fancy version is called the *Fancy Three-Line Sort*.

If you feed the computer the Fancy Three-Line Sort (instead of the Short Three-Line Sort), the computer encounters the "IF X$(I)>X$(I+1)" less often. For the Fancy Three-Line Sort, the number of encounters is $N^2/2$ (instead of $N^3/8$):

Number of names	Encounters ($N^2/2$)	Time
10	50	.5 seconds
12	72	6.72 seconds
20	200	2 seconds
40	800	8 seconds
100	5,000	50 seconds
1,000	500,000	5,000 seconds = 1.4 hours
10,000	50,000,000	500,000 seconds = 5.8 days

Now you can hire the 70-year-old programmer: to alphabetize 10,000 names, he'll need just 5.8 days instead of 39.6 years. By changing just the end of line 60, we saved almost 40 years of computer time—and the programmer's job!

To reduce the time even further, use the *Super-Fancy Three-Line Sort* instead, which is:

```
50 FOR I = 1 TO N-1
60     FOR J = I TO 1 STEP -1: IF X$(J)>X$(J+1) THEN SWAP X$(J),X$(J+1): NEXT
70 NEXT I
```

When typing the Super-Fancy Three-Line Sort, make sure you type line 60 correctly: you must put the IF on the same line as the NEXT. In line 70, make sure you put the "I" after NEXT; otherwise, the computer might think you mean NEXT J. For the Super-Fancy Three-Line Sort, the number of times the computer encounters the phrase "IF X$(J)>X$(J+1)" is only $N^2/4$:

Number of names	Encounters ($N^2/4$)	Time
10	25	.25 seconds
12	36	.36 seconds
20	100	1 second
40	400	4 seconds
100	2,500	25 seconds
1,000	250,000	2,500 seconds = 41.7 minutes
10,000	25,000,000	250,000 seconds = 2.9 days

To reduce the time even further, use the *Six-Line Sort* instead. To construct the Six-Line Sort, begin with the Super-Fancy Three-Line Sort, but change each +1 to +D, and change each -1 to -D, so that you get:

```
50 FOR I = 1 TO N-D
60     FOR J = I TO 1 STEP -D: IF X$(J)>X$(J+D) THEN SWAP X$(J),X$(J+D): NEXT
70 NEXT I
```

D begins by being N—

```
42 D=N
```

but then decreases, so that it becomes a fifth of its original value:

```
44 D=INT(D/5)+1
```

After performing lines 50-70 for that D, check whether D is down to 1 yet; if it isn't, repeat the process:

```
75 IF D>1 THEN GO TO 44
```

So altogether, the complete Six-Line Sort looks like this:

```
42 D=N
44 D=INT(D/5)+1
50 FOR I = 1 TO N-D
60     FOR J = I TO 1 STEP -D: IF X$(J)>X$(J+D) THEN SWAP X$(J),X$(J+D): NEXT
70 NEXT I
75 IF D>1 THEN GO TO 44
```

For the Six-Line Sort, the number of times the computer encounters the phrase "IF X$(J)>X$(J+D)" is only (3/2)*(N-1)↑(4/3):

Number of names	Encounters	Time
10	28	.28 seconds
12	37	.37 seconds
20	76	.76 second
40	198	1.98 seconds
100	687	6.87 seconds
1,000	14,980	149.8 seconds = 2.5 minutes
10,000	323,122	3,231.22 seconds = 53.9 minutes

If the number of names is 10,000, the Six-Line Sort runs much faster than the Super-Fancy Three-Line Sort: the Six-Line Sort takes 53.9 minutes instead of 2.9 days. But if the number of names is just 10, the Six-Line Sort does *not* run faster than the Super-Fancy Three-Line Sort. So use the Six Line Sort only if the number of names is large.

To make the Six-Line Sort even faster, add this line:

```
1 DEFINT A-Z
```

That tells the computer that none of the variables stands for a decimal. By saying DEFINT A-Z, you enable the computer to run the program 20% faster, so that the timing looks like this:

Number of names	Time
10	.2 seconds
12	.3 seconds
20	.6 second
40	1.6 seconds
100	5.5 seconds
1,000	120 seconds = 2 minutes
10,000	2,580 seconds = 43 minutes

We've sure come a long way! Our first attempt (the Short Three-Line Sort) took 39.6 years to alphabetize 10,000 names; our newest attempt (the Six-Line Sort with DEFINT) takes just 43 minutes.

Although I "invented" all those sorting methods, most of them are just slight improvements on methods that were developed by others. For example, the Super-Fancy Three-Line Sort is a slight improvement on the *Shuttle Sort*, which was invented by Shaw & Trimble in 1983. The Six-Line Sort is a slight improvement on the *Shell Sort*, which was invented by Donald Shell in 1959 and further developed by Hibbard & Boothroyd in 1963, Peterson & Russell in 1971, and Knuth in 1973.

If you try running those sorting methods on your own computer, you'll find the timings are slightly different, since the exact timings depend on which computer you have and which names you're alphabetizing.

Problem: write a program that alphabetizes a phone directory.

Solution: suppose you want to alphabetize this little phone directory. . . .

Name	Phone number
Mary Smith	277-8139
John Doe	513-9134
Russ Walter	266-8128
Information	555-1212

Use one of the alphabetizing programs. Type the DATA like this:

```
10 DATA SMITH MARY 277-8139,DOE JOHN 513-9134
20 DATA WALTER RUSS 266-8128,INFORMATION 555-1212
```

The computer will print:

```
DOE JOHN 513-9134
INFORMATION 555-1212
SMITH MARY 277-8139
WALTER RUSS 266-8128
```

Problem: write a program that puts a list of numbers into increasing order. For example, if the numbers are 51, 4.257, -814, 62, and .2, make the computer print. . . .

```
-814
 .2
 4.257
 51
 62
```

Solution: use one of the alphabetizing programs; but in the DATA statement, put the numbers instead of strings; and remove the dollar signs (say X instead of X$).

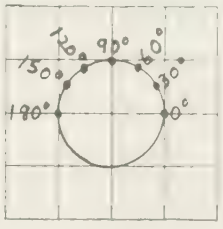
Problem: write a program that puts a list of numbers into *decreasing* order.

Solution: write a program that puts the list of numbers into increasing order; but in line 60, change ">X" to "<X".

ADVANCED MATH

TRIGONOMETRY

On a piece of graph paper, draw a circle, whose center is the graph's origin, and whose radius is 1. Using a protractor, mark the angles on the circle, by putting 0° at the right, and going counterclockwise:



The *sine* of an angle is its Y value. For example:

the sine of 0° is 0
the sine of 30° is .5
the sine of 60° is about .87
the sine of 90° is 1
the sine of 120° is about .87
the sine of 150° is .5
the sine of 180° is 0

The *cosine* of an angle is its X value. For example:

the cosine of 0° is 1
the cosine of 30° is about .87
the cosine of 60° is .5
the cosine of 90° is 0
the cosine of 120° is -.5
the cosine of 150° is about -.87
the cosine of 180° is -1

The *tangent* of an angle is its sine divided by its cosine. For example, the tangent of 60° is about .87/.5, which is 1.74.

This program prints the sine, cosine, and tangent of 60°:

```
10 D=ATN(1)/45
20 PRINT SIN(60*D)
30 PRINT COS(60*D)
40 PRINT TAN(60*D)
```

Exception: to make Atari understand TAN, you must buy the Microsoft BASIC cartridge.

Line 10 is a special formula that defines D to mean "degrees". Line 20 prints the sine of 60 degrees. Line 30 prints the cosine. Line 40 prints the tangent.

In line 20 of that program, the "*D" is important: it tells the computer that you want the sine of 60 *degrees*.

If you omit the "*D", the computer will print the sine of 60 *radians* instead. A radian is larger than a degree. A radian is about 57.3 degrees. More precisely, a radian is $180/\pi$ degrees.

The opposite of the tangent is called the *arctangent*:

the tangent of 60° is about 1.74
the arctangent of 1.74 is about 60°

Similarly, the opposite of the sine is called the *arcsine*, and the opposite of the cosine is called the *arccosine*.

This program prints the arctangent of 1.74, the arcsine of .87, and the arccosine of .5:

```
10 D=ATN(1)/45
20 PRINT ATN(1.74)/D
30 X=.87: PRINT ATN(X/SQR(1-X↑2))/D
40 X=.5: PRINT 90-ATN(X/SQR(1-X↑2))/D
```

Line 10 is the special formula that defines "D" to be "degrees". Line 20 prints the arctangent of 1.74, in degrees. (If you omit the "/D", the computer will print the answer in radians instead of degrees.) Line 30 sets X equal to .87 and then prints its arcsine (by using a formula that combines ATN with SQR). Line 40 sets X equal to .5 and then prints its arccosine (by using a formula that combines 90 with ATN and SQR). The answer to each of the three problems is about 60 degrees.

EXP

e is a special number, which is approximately 2.718281828459045. You can memorize that number easily, if you pair the digits:

2.7 18 28 18 28 45 90 45

That weird number is important in calculus, radioactivity, biological growth, and other areas of science. It's calculated by this formula:

$$e = 1 + \frac{1}{1} + \frac{1}{1*2} + \frac{1}{1*2*3} + \frac{1}{1*2*3*4} + \frac{1}{1*2*3*4*5} + \frac{1}{1*2*3*4*5*6} + \ldots$$

$$= 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \frac{1}{120} + \frac{1}{720} + \ldots$$

EXP(X) means e^X. For example, EXP(3) means e^3, which is e*e*e, which is 2.718281828459045*2.718281828459045*2.718281828459045. EXP(4) means e^4, which is e*e*e*e. EXP(3.1) means $e^{3.1}$, which is more than e^3 but less than e^4.

Here's a practical application. Suppose you put $732 in a savings account, and the bank promises to give you 5% annual interest "compounded continuously". How much money will you have at the end of the year? The answer is 732*EXP(.05).

LOGARITHMS

Here are some powers of 2:

X	2^X
1	2
2	4
3	8
4	16
5	32
6	64

To compute the logarithm-base-2 of a number, find the number in the right column; the answer is in the left column. For example, the logarithm-base-2 of 32 is 5. The logarithm-base-2 of 15 is slightly less than 4. The logarithm-base-2 of 64 is 6. That fact is written:

$\log_2$ 64 is 6

To make the computer find the logarithm-base-2 of 64, say:

```
PRINT 1/LOG(2)*LOG(64)
```

The computer will print 6.

Here are some powers of 10:

X	10^X
1	10
2	100
3	1000
4	10000
5	100000

The logarithm-base-10 of 100000 is 5. The logarithm base-10 of 1001 is slightly more than 3. The logarithm-base-10 of 10000 is 4. That fact is written:

log_{10} 10000 is 4

To make the computer do that calculation, say:

```
PRINT 1/LOG(10)*LOG(10000)
```

The computer will print 4.

The logarithm-base-10 is called the *common logarithm*, and is the kind of logarithm used in high school and chemistry. So if you're studying chemistry and your textbook tells you to find the logarithm of 10000, the textbook means the logarithm-*base-10* of 10000, which is 1/LOG(10)*LOG(10000).

What happens if you forget the base, and say just LOG(10000) instead of 1/LOG(10)*LOG(10000)? If you say just LOG(10000), the computer will find $\log_e$ 10000 (where e is 2.718281828459045), which isn't what your chemistry textbook wants.

LOG(10000), which means $\log_e$ 10000, is called the *natural logarithm* of 10000.

SIGN

If a number is negative, its *sign* is -1. For example, the sign of -546 is -1.

If a number is positive, its *sign* is +1. For example, the sign of 8231 is +1.

The *sign* of 0 is 0.

The computer's abbreviation for "sign" is "SGN". So if you say—

```
PRINT SGN(-546)
```

the computer will print the sign of -546; it will print -1.

If you say—

```
PRINT SGN(8231)
```

the computer will print the sign of 8231; it will print 1.

If you say—

```
PRINT SGN(0)
```

the computer will print the sign of 0; it will print 0.

SGN is the opposite of ABS. Let's see what both functions do to -7.2. ABS removes the minus sign, but leaves the digits:

ABS(-7.2) is 7.2

SGN removes the digits, but leaves the minus sign:

SGN(-7.2) is -1

The Latin word for *sign* is *signum*. Most mathematicians prefer to talk in Latin—they say "signum" instead of "sign"—because the English word "sign" sounds too much like "sine". So mathematicians call SGN the "signum" function.

ATARI BASIC

DIFFERENCES

Recently, many people have been buying the Atari computer. Unfortunately, the Atari computer is quite different from most other microcomputers. Here are the differences....

Getting into BASIC is easy: when you turn the computer on, it will automatically say READY.

To cancel the last character you typed, press the DELETE BACKS key.

To give an abortion, press the BREAK key.

To print on paper (instead of on the screen), say LPRINT (instead of PRINT). To list on paper, say LIST "P:".

For numbers, the safe range goes from 1E-98 to 1E98.

Most *other* computers print a blank space after every number, and a blank space before every number that's positive. But the Atari does *not*.

Like the Nova, the Atari requires you to say DIM, for every string variable.

The Atari does *not* permit the INPUT statement to contain a prompt. Instead of saying—

```
10 INPUT "WHAT IS YOUR NAME";N$
```

you must say:

```
10 PRINT "WHAT IS YOUR NAME";
11 INPUT N$
```

To give a program the name "JOE", and copy it onto the disk, type this:

```
SAVE "D:JOE"
```

To prove the program is on the disk, type the word DOS, then press the RETURN key, then type the letter A, then press the RETURN key twice. The computer will print a catalog of all the programs on disk. Then press the SYSTEM RESET button.

To copy JOE from the disk to the main memory, type this:

```
LOAD "D:JOE"
```

To erase JOE from the disk, type this:

```
DOS
D
JOE
Y
```

The computer will delete JOE. Then press the SYSTEM RESET button.

To replace JOE, say SAVE "D:JOE".

To copy a program onto a tape, adjust the tape recorder, then type the word LPRINT, then press the RETURN key (which might make the computer print an error message, which you can ignore), then type the word CSAVE, then press the RETURN key twice. To retrieve that program, adjust the tape recorder, then type the word CLOAD, then press the RETURN key twice.

To list just line 20, say LIST 20. To list from line 20 to line 50, say LIST 20,50. To list from line 20 to the end, say LIST 20,32767. To list from the beginning to line 20, say LIST 1,20.

To freeze the screen— or to "unfreeze" the screen— do this: while you hold down the CONTROL key, tap the 1 key.

If a FOR...NEXT loop contains a GO TO statement that makes the computer skip out of the loop, you should say "POP: GO TO" instead of "GO TO". If the GO TO statement makes the computer skip out of *two* loops at once, you should say "POP: POP: GO TO" instead of "GO TO".

Instead of saying RND(5), say INT(1+5*RND(0)). Omit the line that says RANDOM.

The screen consists of 4 zones; the width of each zone is 10 characters (except that the width of the rightmost zone is 8 characters); so the width of the entire screen is 38 characters.

A READ statement must not contain a subscript. So instead of saying—

```
READ X(I)
```

say:

```
READ A: X(I)=A
```

The same problem occurs with INPUT; instead of saying—

```
INPUT X(I)
```

say:

```
INPUT A: X(I)=A
```

You can't create a list of strings. You can't create a table of strings.

HOW THE COMPUTER GRIPES

If you have an Atari computer, you can make these 21 major errors:

Error number	Meaning
1	You said to use some equipment. But the equipment's power isn't on.
2	You said to do something. But the computer doesn't have enough memory.
3	You mentioned a number. But the number is too low— or too high.
4	You mentioned a variable. But your program contains too many variables.
5	You mentioned a string. But the string is longer than the DIM statement said.
6	You said READ. But the computer can't find enough DATA to read.
7	You mentioned a number. But that number ought to be between 1 and 32768.
8	You tried to input a string. But the INPUT statement said you'd input a number.
9	You mentioned a subscript. But the subscript is too big or is inconsistent.
10	You gave a long list of GOSUBs, or requested a long computation. But it's too long.
11	You said to compute. But the answer is a number that's too large or too small.
12	You said to go to a different line. But the computer can't find that line.
13	You said NEXT. But the computer can't find the FOR that matches the NEXT.
14	You gave a long instruction. But the instruction is too long.
15	You said NEXT or RETURN. But you erased the corresponding FOR or GOSUB.
16	You said RETURN. But the computer can't find the GOSUB that matches the RETURN.
17	You told the computer to do something. But the computer's memory contains an error.
18	You mentioned a string. But one of the characters in the string is inappropriate.
19	You said to LOAD a long program. But the computer doesn't have enough memory.
20	You mentioned a device number. But the number is larger than 7.
21	You said to LOAD. But the computer can't load that kind of file.

For example, suppose line 50 says to do something, but the computer doesn't have enough memory. The computer will say:

```
ERROR-  2 AT LINE 50
```

Atari's computer contains four *voices*; each voice can sing. So you can make the computer imitate a quartet.

The four voices are called 0, 1, 2, and 3. This program makes them sing:

```
10 SOUND 0,127,10,4
20 SOUND 1,101,10,4
30 SOUND 2,84,10,4
40 SOUND 3,63,10,4
50 FOR I = 1 TO 400: NEXT I

60 SOUND 3,56,10,4
70 FOR I = 1 TO 400: NEXT I
```

Line 10 tells voice 0 to sing pitch 127 (which is "middle C"), using tone quality 10 (which is "pipe organ") and volume 4 (which is "medium-soft"). Line 20 tells voice 1 to sing pitch 101 (which is "the E above middle C"). Line 30 tells voice 2 to sing pitch 84 (which is "the G above middle C"). Line 40 tells voice 3 to sing pitch 63 (which is "the C above middle C"). So line 10-40 make all four voices sing; you hear a chord. Line 50 makes the computer pause for 1 second; while the computer pauses, the voices continue to sing the chord; so you hear the chord for 1 second.

Line 60 tells voice 3 to change its pitch to 56 (which is "the D above the C above middle C"); the other voices, unaffected, continue to sing C, E, and G. Line 70 makes the computer pause, so that you hear the new chord for 1 second.

Pitch

The pitch must be a number from 0 to 255, and cannot be a decimal. Here are some popular pitches:

255 is a deep bass note (it's "the C below middle C"; "the C in the bass clef")
127 is "middle C"
63 is a soprano note ("the C above middle C"; "the C in the treble clef")
31 is a very high note ("the C above the treble clef")
15 is very, very high ("the C above the C above the treble clef")
7 is very, very, very high ("the C above the C above the C above the treble clef")
3 is very, very, very, very high ("a C at the edge of your range of hearing")
1 is very, very, very, very, very high ("a C that only your dog can hear")
0 is very, very, very, very, very, very high ("maybe not even your dog?")

To translate traditional music notation into computer pitches, use this chart:

	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
bass	255	241	227	214	202	191	180	170	160	151	143	135
alto	127	120	113	107	101	95	90	84	80	75	71	67
soprano	63	59	56	53	50	47	44	42	39	37	35	33
very high	31	29	28	26	24	23	22	20	19	18	17	16
very, very high	15	14	13		12	11	10		9		8	
very, very, very high	7		6			5			4			
edge of range of hearing	3					2						
dogs only	1											
not even dogs?	0											

(Note to musicians: to make full use of the Atari's range, I've tuned it slightly flat. For example, 127, which I'm calling "middle C", is slightly flatter than a piano's middle C; in fact, it's closer to the piano's B.)

Tone quality

The tone quality must be 0, 2, 4, 6, 8, 10, 12, or 14. For most music, use 10. The 14 sounds almost the same as 10. The 12 is a buzz. The other tones are used for making the computer imitate guns, bombs, airplanes, trucks, motorcycles, and all the other evil joys of life.

Volume

The volume must be from 0 to 15. The 0 is silence; 1 is a barely audible whisper; 4 is soft and sweet and is what I recommend for must four-part harmony. The higher the volume, the more unpleasant distortion you'll notice— especially if you try harmony

My favorite composition

This program composes exciting music:

```
10 FOR I = 0 TO 255
20      SOUND 0,I,10,4
30      SOUND 1,255-I,10,4
40      SOUND 2,INT(RND(0)*256),10,4
50 NEXT I
```

Line 10 says that I increases. As I increases from 0 to 255, the pitch in line 20 goes from 0 to 255; the pitch in line 30 goes from 255 to 0; and the pitch in line 40 is random. So voice 0 goes from 0 (dogs only) to 255 (bass), which voice 1 goes from 255 (bass) to 0 (dogs only), and voice 2 sings random notes. Try it!

GRAPHICS

On Atari's computer, you can use many graphics modes, which are numbered from 0 to 24. Your program's top line must say which graphics mode to use. For example, if you wish to use graphics mode 19, your program's top line must say:

```
10 GRAPHICS 19
```

Here's how to use each popular graphics mode....

Mode 19. In this mode, the X coordinate must be from 0 to 39, and the Y coordinate must be from 0 to 23. Initially, the entire screen is very, very dark gray (i.e., black). You can use these colors:

COLOR 0 *is black.*
COLOR 1 *is medium-light orange.*
COLOR 2 *is light green.*
COLOR 3 *is dark blue-turquoise.*

Example:

```
Use graphics mode 19. Make the entire screen black.   10 GRAPHICS 19
Make pixel 27,13 and pixel 25,19 be very light green. 20 COLOR 2: PLOT 27,13: PLOT 25,19
Draw a medium-light orange line from 20,17 to 5,2.    30 COLOR 1: PLOT 20,17: DRAWTO 5,2
Pause, to let the human admire the picture.           40 GO TO 40
```

To abort that program, and to stop using graphics mode 19, press the SYSTEM RESET button.

Mode 21. This resembles mode 19. But the X coordinate can go from 0 to 79, and the Y coordinate can go from 0 to 47.

Mode 23. This resembles mode 19. But the X coordinate can go from 0 to 159, and the Y coordinate can go from 0 to 95.

Mode 20. This resembles mode 21, but lacks COLOR 2 and COLOR 3, and therefore consumes less memory.

Mode 22. This resembles mode 23, but lacks COLOR 2 and COLOR 3, and therefore consumes less memory.

Mode 24. To use this mode, you must buy at least 16K of RAM memory. In this mode, the X coordinate can go from 0 to 319; the Y coordinate can go from 0 to 191. Initially, the entire screen is dark blue-turquoise. You can use these colors:

COLOR 0 *is dark blue-turquoise.*
COLOR 1 *is light blue-turquoise.*

Example:

```
Use graphics mode 24. Make the screen dark. 10 GRAPHICS 24
Draw a light line from 20,7 to 302,146.     20 COLOR 1: PLOT 20,7: DRAWTO 302,146
Pause, to let the human admire the picture. 30 GO TO 30
```

To abort that program, and to stop using graphics mode 24, press the SYSTEM RESET button.

Mode 0. This is the "normal" mode: it's the mode you find yourself in, whenever you press SYSTEM RESET, and whenever you aren't thinking about graphics. In this mode, the X coordinate can go from 0 to 39, and the Y coordinate can go from 0 to 23. Initially, the entire screen is dark greenish-blue— except for the screen's border, which is black. Example:

```
Use graphics mode 0. Make screen dark, border black. 10 GRAPHICS 0
Beginning at pixel 14,9, print the word JAR.         20 POSITION 14,9: PRINT "JAR";
Pause, to let the human admire it.                   30 GO TO 30
```

That program prints the word JAR, so that the J is at pixel 14,9, and so that the A is at the next pixel (15,9), and so that the R is at the next pixel (16,9). The word JAR is light greenish-blue.

Mode 17. In this mode, the X coordinate must be from 0 to 19, and the Y coordinate must be from 0 to 23. Initially, the entire screen is black. Example:

```
Use graphics mode 17. Make the entire screen black. 10 GRAPHICS 17
Beginning at pixel 14,9, print the word JAR.        20 POSITION 14,9: PRINT#6; "JAR";
Pause, to let the human admire it.                  30 GO TO 30
```

That program prints the word JAR, so that the J is at pixel 14,9, and so that the A is at the next pixel (15,9), and so that the R is at the next pixel (16,9). The word JAR is medium-light orange. If you try to print a *small* (lower-case) letter (by pressing the LOWR key), the computer will print it capitalized instead, and will make it light green. If you try to print a *dark on light* capital letter (by pressing the ⅄ key), the computer will instead make it dark blue-turquoise on a black background. If you try to print a *dark on light small* letter (by pressing the ⅄ and LOWR keys), the computer will instead make it a medium-dark red capital on a black background.

Mode 18. This resembles mode 17. But the Y coordinate must go from 0 to 11.

Split-screen modes. You can make the bottom part of the screen be in mode 0, and the top part of the screen be in a different mode. Here's how:

Graphics mode	Meaning
GRAPHICS 1	The screen's bottom is in mode 0; the screen's top is in mode 17.
GRAPHICS 2	The screen's bottom is in mode 0; the screen's top is in mode 18.
GRAPHICS 3	The screen's bottom is in mode 0; the screen's top is in mode 19.
etc.	
GRAPHICS 8	The screen's bottom is in mode 0; the screen's top is in mode 24.

In those split-screen modes, the bottom sixth of the screen is in mode 0; the top five-sixths are in a mode from 17 to 24. The bottom sixth, which is in mode 0, contains just 4 lines of text. To create that text, you can say PRINT, but you can't say POSITION: in split-screen mode, the bottom sixth doesn't understand POSITION. The top five-sixths, which are in a mode from 17 to 24, have fewer Y coordinates than if the whole screen were devoted to that mode: a sixth of the Y coordinates are missing.

Changing the colors. You've seen that Atari's computer normally uses these colors:

Color	Major use
black	mode 19's "color 0"; mode 17's background; mode 0's border
medium-light orange	mode 19's "color 1"; mode 17
light green	mode 19's "color 2"; mode 17
dark blue-turquoise	mode 19's "color 3"; mode 17; mode 0; mode 24's "color 0"
medium-dark red	mode 17
light blue-turquoise	mode 17; mode 0; mode 24's "color 1"

Each of those colors has a special *register number*:

Color	Register number
black	4
medium-light orange	0
light green	1
dark blue-turquoise	2
medium-dark red	3
light blue-turquoise	1

Atari also uses hue and luminance numbers:

Hue	Number
gray	0
gold	1
orange	2
red-orange	3
red	4
red-purple	5
purple	6
blue-purple	7
blue	8
blue-turquoise	9
turquoise	10
green-turquoise	11
green	12
green-russet	13
russet-green	14
russet	15

Luminance	Number
very, very dark	0
very dark	2
dark	4
medium-dark	6
medium-light	8
light	10
very light	12
very, very light	14

If you say—

```
SETCOLOR 4,1,10
```
register hue luminance

the computer will replace black (which is register 4) by a gold (hue 1) that's light (luminance 10). Everything on the screen that was black suddenly becomes light gold; and whenever you tell the computer to do further drawing in black, it will draw light gold instead. To make the computer stop turning black into light gold, give a SETCOLOR command that contradicts it, or hit the SYSTEM RESET button, or change to a different GRAPHICS mode.

Here's another example. If you say—

```
SETCOLOR 2,4,6
```
register hue luminance

the computer will replace dark blue turquoise (which is register 2) by a red (hue 4) that's medium-dark. Everything on the screen that was dark blue-turquoise suddenly becomes medium-dark red.

In a typical Atari graphics program, line 10 says GRAPHICS and gives the GRAPHICS mode; line 20 says SETCOLOR and changes a boring register color (such as black) to a more interesting color (such as light gold); lines 30, 40, and 50 contain additional SETCOLOR statements.

Register 1½ automatically has the same luminence as register 1, and the same hue as register 2. (Since register 1's luminence is normally "light", and since register 2's hue is normally "blue-turquoise", register 1½ is normally "light blue-turquoise".) To change register 1½'s color from light blue turquoise to a different color, change registers 1 and 2, by saying SETCOLOR 1 and SETCOLOR 2. Do *not* try to say SETCOLOR 1½.

In graphics mode 24, the pixels are very tiny and hard to see. To make them easier to see, make them black on a white background, by saying:

```
10 GRAPHICS 24
20 SETCOLOR 2,0,14  changes "dark blue-turquoise" to "very, very light gray" (white)
30 SETCOLOR 1,0,0   changes register 1½ to black
```

TRICKY TYPING

TWO SHIFT KEYS

Your keyboard probably has two SHIFT keys. (Exception: the TS 1000, TS 1500, and TI 99/4A computers are inferior and have only one SHIFT key.)

One SHIFT key is on the left side of the keyboard; the other SHIFT key is on the right side. It doesn't matter which SHIFT key you press. Press whichever SHIFT key makes your fingers comfortable.

LOCK KEY

On most keyboards, you'll see a key that says LOCK (or CAPS LOCK or CAPS or SHIFT LOCK or SHIFT). Here's how to use it.

Such a keyboard can operate in two modes: "upper-case only" and "upper-and-lower-case". If it's operating in "upper-case only", all the letters are automatically capitalized. If it's operating in "upper-and-lower-case", the letters are *not* automatically capitalized: to create a capital letter, you must press the SHIFT key.

To switch from one mode to the other, tap the LOCK key. The keyboard will stay in the new mode, until you tap the LOCK key again.

The LOCK key affects only the capitalization of *letters*. It does *not* affect numbers or punctuation.

For example, the LOCK key has no effect on the [$ / 4] key. Regardless of which mode you're locked into, pressing the [$ / 4] key produces a 4 (unless you press the SHIFT key).

Some computers are abnormal. For example, on Atari 400 and 800 computers, to switch to upper-case-only you must tap the LOCK key *while holding down the SHIFT key*. On Radio Shack's model 3 computer, the LOCK key is missing: instead of tapping a LOCK key, tap the zero key while holding down the SHIFT key. On Coleco's Adam computer, the LOCK key *does* affect numbers and punctuation.

NUMERIC KEYPAD

On the far right side of your keyboard, you might see an extra cluster of keys that contain numbers. That group of keys is called the *numeric keypad*. For example, Radio Shack's numeric keypad looks like this:

7	8	9
4	5	6
1	2	3
0	.	ENTER

To type a number, use the numeric keypad...or use the top row of the main keyboard. Use whichever makes your fingers comfortable.

CANCELING A LINE

If you've been typing a line and regret it, here's how to cancel the entire line:

Computer	How to cancel the line you've been typing
CDC, PDP-8, HP-2000	Press the ESCAPE key.
IBM maxi (using CMS)	Type a cent sign (¢).
Nova, Eclipse	While you hold down the SHIFT key, tap the L key.
Radio Shack	While you hold down the SHIFT key, tap the left-arrow (←).
Atari	While you hold down the SHIFT key, tap the DELETE key.
PDP-10, PDP-11, PDP-20	While you hold down the CONTROL key, tap the U key.
Apple, HP-3000, Honeywell	While you hold down the CONTROL key, tap the X key.
TI 99/4A	While you hold down the FCTN key, tap the 3 key.

REVERSING THE COLOR

Instead of always printing light characters on a dark background, some computers let you reverse the colors, so that you see *dark* characters on a *light* background.

This program prints the word UNBEATABLE, so that the UN is normal (light-on-dark), the BEAT is reversed (dark-on-light), and the ABLE is normal again (light-on-dark):

Apple

```
10 PRINT "UN";

20 INVERSE
30 PRINT "BEAT";

40 NORMAL
50 PRINT "ABLE"
```

Atari

```
10 PRINT "UNBEATABLE"
```

Here, press the [key] key. Here, press the [key] key.

VIC-20, Commodore 64

```
10 PRINT "UNBEATABLE"
```

Here, while holding down the CTRL key, hit RVS ON key. Here, while holding down the CTRL, hit RVS OFF.

BLINKING

If you have an Apple computer, you can make it blink. This program prints the word UNBEATABLE, so that the UN is normal (light-on-dark), the BEAT is flashing (dark-on-light, alternating with light-on-dark), and the ABLE is normal again (light-on-dark):

```
10 PRINT "UN";

20 FLASH
30 PRINT "BEAT";

40 NORMAL
50 PRINT "ABLE"
```

FAT CHARACTERS

If you have Radio Shack's model 3 or 4, each character on the screen normally looks tall and thin. To make the characters look fatter, tap the key that has a right-arrow (→), while holding down the SHIFT key. Presto, the characters on your screen become fatter! Henceforth, all characters you type will be fat...until you clear the screen (by pressing the CLEAR key, or typing CLS or NEW).

A different way to make the characters fatter is to give this command:

```
PRINT CHR$(23);
```

Here's a sample program:

```
10 CLS                                   Erase the screen, and make sure the characters are thin.
20 PRINT "FUN"                           Print the word FUN, using thin characters.
30 FOR I = 1 TO 400: NEXT                Pause 1 second, to make the human stare at the word FUN.

40 CLS                                   Erase the screen.
50 PRINT CHR$(23);                       Change to fat-character mode.
60 PRINT "SURPRISE"                      Print the word SURPRISE, using fat characters.
70 FOR I = 1 TO 400: NEXT                Pause 1 second, to make the human stare at SURPRISE.

80 GO TO 10                              Repeat.
```

HOW TO SAY "IF"

Let's make the computer print a question mark and wait for the human to type a number. If the human types a number less than 4, let's make the computer say THE NUMBER IS SMALL. If the number is *not* less than 4, let's make the computer say THE NUMBER IS LARGE.

To write the program, you can use four methods:

Method 1
```
10 INPUT N
20 IF N<4 THEN PRINT "THE NUMBER IS SMALL" ELSE PRINT "THE NUMBER IS LARGE"
```

Method 2
```
10 INPUT N
20 IF N<4 THEN PRINT "THE NUMBER IS SMALL"
30 IF N>=4 THEN PRINT "THE NUMBER IS LARGE"
```

Method 3
```
10 INPUT N
20 IF N<4 THEN PRINT "THE NUMBER IS SMALL": END
30 PRINT "THE NUMBER IS LARGE"
```

Method 4
```
10 INPUT N
20 IF N<4 THEN 100
30 PRINT "THE NUMBER IS LARGE"
40 STOP
100 PRINT "THE NUMBER IS SMALL"
```

Method 1 is the briefest and the best. In that method, line 20 means:

If N is less than 4, print THE NUMBER IS SMALL;
otherwise, print THE NUMBER IS LARGE.

So it means:

Print THE NUMBER IS SMALL, or else print THE NUMBER IS LARGE— depending on whether N is less than 4.

Method 2 is more long-winded, but is still easy to understand. Its lines 20 and 30 mea

If N is less than 4, print THE NUMBER IS SMALL.
If N is greater than or equal to 4, print THE NUMBER IS LARGE.

Method 3 resembles method 2, but is briefer: it says "END" instead of "IF N>=4 THEN". If N is less than 4, the computer will print THE NUMBER IS SMALL and then end. If N is *not* less than 4, the computer will ignore line 20, and will therefore reach line 30, which prints THE NUMBER IS LARGE.

Method 4 is the most long-winded, the ugliest, the hardest to read, and the worst. Its line 20 means:

If N is less than 4, then go to line 100.

I recommend that you use method 1, if possible. But some computers don't understan the word ELSE, and therefore don't understand method 1; on such computers, you must use one of the other methods.

Method 3 contains a colon in line 20. Some computers get confused by that colon. On such computers, you must pick a different method instead.

Method 4 is ugly, but has one advantage: it works on *all* computers. If you're trying to write a program that will work on *all* computers, you must use method 4. If your computer is so stupid that it doesn't understand methods 1, 2, or 3, you must use method 4. Otherwise, avoid method 4 like the plague— because it will confuse you so badly that you'll make lots of errors.

If you're buying a computer, try to buy one that understands all four methods— or at least understands methods 2, 3, and 4. If you buy a computer that understands just method 4, you're asking for trouble!

Here's how various computers fare:

Computer	What the computer understands
IBM PC, DEC Rainbow, CP/M, Radio Shack, Atari using Microsoft	The computer understands all four methods.
TI 99/4A using Extended BASIC	The computer understands all four methods— but use a double colon (::) instead of a colon (:).
DEC mini or maxi using BASIC-PLUS-2	The computer understands all four methods— but use a backslash (\) instead of a colon (:), and say STOP instead of END.
DEC mini using BASIC-PLUS	The computer understands all four methods— but use a backslash (\) instead of a colon (:), and say STOP instead of END. Warning: if you try to put a backslash between the words THEN and ELSE, the computer will get confused and gripe.
Apple, Commodore, Atari without Microsoft	The computer doesn't understand the word ELSE. Therefore, you can't use method 1. The other three methods all work.
ZX Spectrum, TS 2068	The computer doesn't understand the word ELSE. Therefore, you can't use method 1. The other three methods all work— but say STOP instead of END.
IBM maxicomputers	The computer doesn't understand the colon. Therefore, you can't use method 3. The other methods all work, for this example. (If the example were more complicated, so that method 1 would require a colon, method 1 wouldn't work.)
North Star using North Star's BASIC instead of CP/M	If you put a colon in an IF statement, the computer will handle the IF statement incorrectly. (The computer will think the IF statement ends at the colon.) Therefore, do *not* put a colon in an IF statement; do *not* use method 3, which doesn't work. The other methods all work, for this example. (If the example were more complicated, so that method 1 would require a colon, method 1 wouldn't work.)
Nova, Eclipse	The computer doesn't understand the word ELSE, and doesn't understand the colon. Therefore, the only methods that work are 2 and 4.
ZX-80, ZX-81, TS 1000, TS 1500	The computer doesn't understand the word ELSE, and doesn't understand the colon. Therefore, the only methods that work are 2 and 4. Say STOP instead of END.
TI 99/4A without Extended BASIC	After the word THEN, you must put a line number— as in method 4. The computer understands only method 4. So you must use method 4.
Univac, CDC	After the word THEN, you must put a line number— as in method 4. The computer understands only method 4. So you must use method 4. Say STOP instead of END.

ON...GO TO

In your program, you can say:

```
50 ON N GO TO 80,100,20,350
```

That means: go to either 80, 100, 20, or 350; the decision depends on what N is. In other words:

If N is 1, go to 80.
If N is 2, go to 100.
If N is 3, go to 20.
If N is 4, go to 350.

In that example, if N is less than 1 or greater than 4, what happens? If your computer is a TI 99/4A or Honeywell or CDC or uses DEC's BASIC-PLUS or BASIC-PLUS-2, the computer will gripe. Most other computers, instead of griping, will proceed to the line underneath line 50 (which is probably line 60).

Weird computers

For Atari computers, you must omit the space between GO and TO. For IBM maxicomputers (which all have a worse version of BASIC than the IBM PC!), you must write the statement like this:

```
50 GOTO 80,100,20,350 ON N
```

Sinclair computers don't understand ON...GO TO at all.

Christmas

Remember the Christmas carol called "The Twelve Days of Christmas"? The first three verses go like this:

```
ON THE FIRST DAY OF CHRISTMAS, MY TRUE LOVE SENT TO ME
A PARTRIDGE IN A PEAR TREE.

ON THE SECOND DAY OF CHRISTMAS, MY TRUE LOVE SENT TO ME
TWO TURTLE DOVES...AND
A PARTRIDGE IN A PEAR TREE.

ON THE THIRD DAY OF CHRISTMAS, MY TRUE LOVE SENT TO ME
THREE FRENCH HENS,
TWO TURTLE DOVES...AND
A PARTRIDGE IN A PEAR TREE.
```

This program prints the final verse:

```
10 PRINT "ON THE TWELFTH DAY OF CHRISTMAS, MY TRUE LOVE SENT TO ME"
20 PRINT "TWELVE DRUMMERS DRUMMING,"
30 PRINT "ELEVEN PIPERS PIPING,"
40 PRINT "TEN LORDS A-LEAPING,"
50 PRINT "NINE LADIES DANCING,"
60 PRINT "EIGHT MAIDS A-MILKING,"
70 PRINT "SEVEN SWANS A-SWIMMING,"
80 PRINT "SIX GEESE A-LAYING,"
90 PRINT "FIVE GO---OLD RINGS,"
100 PRINT "FOUR CALLING BIRDS,"
110 PRINT "THREE FRENCH HENS,"
120 PRINT "TWO TURTLE DOVES...AND"
130 PRINT "A PARTRIDGE IN A PEAR TREE."
```

This program prints all twelve verses:

```
→→1 DATA FIRST,SECOND,THIRD,FOURTH,FIFTH,SIXTH,SEVENTH,EIGHTH,NINTH
→→2 DATA TENTH,ELEVENTH, TWELFTH
→→3 FOR V = 1 TO 12
→→4       READ W$
→→10      PRINT "ON THE ";W$;" DAY OF CHRISTMAS, MY TRUE LOVE SENT TO ME"
→→11      ON V GO TO 130,120,110,100,90,80,70,60,50,40,30,20
  20      PRINT "TWELVE DRUMMERS DRUMMING,"
  30      PRINT "ELEVEN PIPERS PIPING,"
  40      PRINT "TEN LORDS A-LEAPING,"
  50      PRINT "NINE LADIES DANCING,"
  60      PRINT "EIGHT MAID A-MILKING,"
  70      PRINT "SEVEN SWANS A-SWIMMING,"
  80      PRINT "SIX GEESE A-LAYING,"
  90      PRINT "FIVE GO---OLD RINGS,"
  100     PRINT "FOUR CALLING BIRDS,"
  110     PRINT "THREE FRENCH HENS,"
  120     PRINT "TWO TURTLE DOVES...AND"
  130     PRINT "A PARTRIDGE IN A PEAR TREE."
→→140     PRINT
→→150 NEXT V
```

Lines 1 and 2 teach the computer how to spell the words "FIRST", "SECOND", "THIRD", etc. Line 3 makes the computer do lines 4 through 140, twelve times (once for each verse). Lines 4 and 10 look at the data and print the top line of the verse. Line 11 says:

If V is 1, go to line 130 (so the computer prints A PARTRIDGE IN A PEAR TREE).

If V is 2, go to line 120 (so the computer prints TWO TURTLE DOVES...AND A PARTRIDGE IN A PEAR TREE).

If V is 3, go to line 110 (so the computer prints THREE FRENCH HENS, TWO TURTLE DOVES...AND A PARTRIDGE IN A PEAR TREE).

And so on, for each verse.

ON...GOSUB

Many computers let you say:

```
50 ON N GOSUB 80,100,20,350
```

That means:

If N is 1, GOSUB 80.
If N is 2, GOSUB 100.
If N is 3, GOSUB 20.
If N is 4, GOSUB 350.

Sinclair, CDC, and Univac computers don't understand ON...GOSUB. The TI 99/4A understands ON...GOSUB only if you buy TI's Extended BASIC cartridge. If your IBM computer is a maxi instead of a PC, you must write the statement like this:

```
50 GOSUB 80,100,20,350 ON N
```

STOP VERSUS END

The word STOP resembles the word END. Here's the difference:

Computer	Difference between STOP and END
TI 99/4A	STOP is exactly the same as END.
Timex Sinclair	The computer doesn't understand END. You must say STOP.
IBM PC, DEC Rainbow, CP/M, Radio Shack, Apple, Commodore, Atari, Data General	STOP is the same as END—except that when the computer reaches the word STOP, it prints a message saying what line it stopped at.
DEC mini or maxi using BASIC-PLUS or BASIC-PLUS-2	You can put the word STOP anywhere in your program. The word END is permitted only on the bottom line. If the computer reaches the word STOP, it prints a message that says what line it stopped at.
Honeywell, Univac, DEC mini or maxi using neither BASIC-PLUS nor BASIC-PLUS-2	The word END *must* appear on the bottom line, and *only* on the bottom line. For other lines, say STOP.
IBM maxicomputers	Say STOP, not END. (If you say END, the computer will ignore everything that comes underneath the END; so END will become the bottom line that the computer will look at.)

ERROR TRAPS

When you run a program, if the computer encounters an error, the computer will want to gripe. But instead of letting the computer gripe, you can force the computer to do something different.

For example, you can tell the computer this: "If you encounter an error in lines 50-70, don't gripe; instead of griping, do the routine in lines 1000-1020 and then go back to line 30 (to try again)." To tell the computer that, insert these lines:

IBM PC, CP/M, DEC Rainbow, Atari using Microsoft BASIC, Radio Shack (except Color Computer)

```
49 ON ERROR GO TO 1000
71 ON ERROR GO TO 0
999 END
1021 RESUME 30
```

DEC mini or maxi, using BASIC-PLUS or BASIC-PLUS-2

```
49 ONERROR GO TO 1000
71 ONERROR GO TO 0
999 STOP
1021 RESUME 30
```

Nova, Eclipse

```
49 ON ERR THEN GO TO 1000
71 ON ERR THEN STOP
999 END
1021 GO TO 30
```

Apple

```
49 ON ERR GO TO 1000
81 POKE 216,0
999 END
1021 GO TO 30
```

Atari without Microsoft BASIC

```
49 TRAP 1000
81 TRAP 40000
999 END
1021 GO TO 30
```

Lines 49 and 71 say: if an error is encountered between lines 49 and 71, do not gripe; instead, go to line 1000. So if an error occurs in lines 50-70, the computer will go to line 1000, and will do lines 1000-1020, then come to line 1021, which sends the computer back to line 30.

Line 999 marks the end of the main routine, and separates the main routine from the subroutine that begins in line 1000. The subroutine is called an *error-handing routine* or an *error trap*. It ends with the word RESUME or GO TO, instead of RETURN.

In line 1071, you can change the 30 to any other number you please; you can send the computer wherever you wish. To send the computer back to the line that contained the error, and make the computer do that line again, change line 1021 to this:

IBM PC, CP/M, DEC, Apple,
Atari using Microsoft BASIC,
Radio Shack (except Color Computer)

```
1021 RESUME
```

Nova, Eclipse

```
1021 RETRY
```

To send the computer back to the line *underneath* the line that contained the error, change line 1021 to this:

IBM PC, CP/M, DEC Rainbow,
Atari using Microsoft BASIC,
Radio Shack (except Color Computer)

```
1021 RESUME NEXT
```

Nova, Eclipse

```
1021 RETURN
```

To make that work properly, the line that contained the error should consist of just one statement (instead of statements separated by colons).

HOW TO SPEED UP THE COMPUTER

These tricks will make the typical microcomputer run more quickly....

Instead of exponents, use multiplication:

SLOW

```
50 Y=X↑2
```

FASTER

```
50 Y=X*X
```

Combine statements, to form a single line:

SLOW

```
50 A=3
60 B=7
```

FASTER

```
50 A=3: B=7
```

Warning: an IF statement cannot be combined with a later statement.

CANNOT BE COMBINED

```
50 IF I<1 THEN 200
60 B=7
```

If a number contains a decimal point and is in a loop, turn the number into a variable:

SLOW

```
50 FOR I = 1 TO 1000
60     S=S+3.1/I
70 NEXT
```

FASTER

```
49 C=3.1
50 FOR I = 1 TO 1000
60     S=S+C/I
70 NEXT
```

Omit the variable after NEXT (unless the FOR...NEXT loop contains another FOR...NEXT loop):

SLOW

```
50 NEXT I
```

FASTER

```
50 NEXT
```

If your program doesn't involve decimals or large numbers, put this statement at the beginning of your program:

```
1 DEFINT A-Z
```

It tells the computer that every numeric variable will be an integer between -32768 and 32767.

If your program involves only a *few* decimals or large numbers, put this statement at the beginning of your program—

```
1 DEFINT A-Z
```

—and put an exclamation point after each variable that stands for a decimal or large number, like this:

```
50 X!=9.5
60 Y!=5931742
70 Z!=X!*3.7
```

The DEFINT statement works only on superior microcomputers, such as the IBM PC, DEC Rainbow, CP/M, Radio Shack models 3, 4, 12, and 16, and Atari using Microsoft BASIC. The following microcomputers are inferior and can't handle the DEFINT statement: Sinclair, TI 99/4A, Radio Shack Color Computer, Commodore, Apple, and Atari without Microsoft BASIC.

DATA FILES

SEQUENTIAL ACCESS

This program makes the computer print a message onto your terminal's screen or paper:

```
10 PRINT "EAT"
20 PRINT 2+2
30 PRINT "EGGS"
```

The computer will print:

```
EAT
 4
EGGS
```

Instead of making the computer print onto your terminal, you can make the computer print onto a disk. For example, if you want the computer to create a file named SUE, on a disk, and if you want SUE to contain this message—

```
EAT
 4
EGGS
```

type this:

DEC using BASIC-PLUS or BASIC-PLUS-2

```
5 OPEN "SUE" FOR OUTPUT AS FILE 1

10 PRINT#1, "EAT"
20 PRINT#1, 2+2
30 PRINT#1, "EGGS"

40 CLOSE 1
```

IBM PC

```
5 OPEN "SUE" FOR OUTPUT AS 1

10 PRINT#1, "EAT"
20 PRINT#1, 2+2
30 PRINT#1, "EGGS"

40 CLOSE 1
```

CP/M, Radio Shack

```
5 OPEN "O",1,"SUE"

10 PRINT#1, "EAT"
20 PRINT#1, 2+2
30 PRINT#1, "EGGS"

40 CLOSE 1
```

Atari using Microsoft

```
5 OPEN#1,"D:SUE" OUTPUT

10 PRINT#1, "EAT"
20 PRINT#1, 2+2
30 PRINT#1, "EGGS"

40 CLOSE#1
```

Atari without Microsoft

```
5 OPEN#1,8,0,"D:SUE"

10 PRINT#1; "EAT"
20 PRINT#1; 2+2
30 PRINT#1; "EGGS"

40 CLOSE#1
```

Commodore

```
5 OPEN 2,8,2,"SUE,S,W"

10 PRINT#2, "EAT"
20 PRINT#2, 2+2
30 PRINT#2, "EGGS"

40 CLOSE 2
```

Apple

```
5 D$=CHR$(4): ?D$"OPEN SUE": ?D$"WRITE SUE"

10 PRINT "EAT"
20 PRINT 2+2
30 PRINT "EGGS"

40 ?D$"CLOSE SUE"
```

Warning: for CP/M and Radio Shack computers, line 5 must contain the letter O, not a zero; the letter O means "Output". For Atari without Microsoft, line 5 must contain a zero, which means "no abnormalities".

When you run that program, the computer won't print anything onto your screen: it will print onto the disk instead.

To see SUE, run this program, which inputs from SUE and prints onto your screen:

DEC using BASIC-PLUS or BASIC-PLUS-2

```
5 OPEN "SUE FOR INPUT AS FILE 1

10 INPUT#1, A$
11 PRINT A$

20 INPUT#1, B
21 PRINT B

30 INPUT#1, C$
31 PRINT C$

40 CLOSE 1
```

IBM PC

```
5 OPEN "SUE" FOR INPUT AS 1

10 INPUT#1, A$
11 PRINT A$

20 INPUT#1, B
21 PRINT B

30 INPUT#1, C$
31 PRINT C$

40 CLOSE 1
```

CP/M, Radio Shack

```
5 OPEN "I",1,"SUE"

10 INPUT#1, A$
11 PRINT A$

20 INPUT#1, B
21 PRINT B

30 INPUT#1, C$
31 PRINT C$

40 CLOSE 1
```

Atari using Microsoft

```
5 OPEN#1,"D:SUE" INPUT

10 INPUT#1, A$
11 PRINT A$

20 INPUT#1, B
21 PRINT B

30 INPUT#1, C$
31 PRINT C$

40 CLOSE#1
```

Atari without Microsoft

```
5 DIM A$(3), C$(4): OPEN#1,4,0,"D:SUE"

10 INPUT#1, A$
11 PRINT A$

20 INPUT#1, B
21 PRINT B

30 INPUT#1, C$
31 PRINT C$

40 CLOSE#1
```

Commodore

```
5 OPEN 2,8,2,"SUE,S,R"

10 INPUT#2, A$
11 PRINT A$

20 INPUT#2, B
21 PRINT B

30 INPUT#2, C$
31 PRINT C$

40 CLOSE 2
```

Apple

```
5 D$=CHR$(4): ?D$"OPEN SUE": ?D$"READ SUE"

10 INPUT A$
11 PRINT A$

20 INPUT B
21 PRINT B

30 INPUT C$
31 PRINT C$

40 ?D$"CLOSE SUE"
```

Line 10 inputs A$ from SUE, so A$ is EAT. Line 11 prints EAT onto your screen.
Line 20 inputs B from SUE, so B is 4. Line 21 prints 4 onto your screen.
Line 30 inputs C$ from SUE, so C$ is EGGS. Line 31 prints EGGS onto your screen.
So on your screen, you'll see EAT and 4 and EGGS.

RANDOM ACCESS

On a disk, you can store two kinds of data files. The simple kind is called a *sequential-access* data file; the complicated kind is called a *random-access* (or *relative-access* or *direct-access*) data file. You've already learned how to create and retrieve a sequential-access data file. Now let's look at random-access.

Though more complicated than *sequential-access* data files, *random-access* data files have an advantage: they let you skip around. In a sequential-access data file, you must look at the first piece of data, then at the second, then at the third, etc. In a random-access data file, you can look at the eighth piece of data, then skip directly to the eleventh, then hop back to the third, then skip directly to the sixth, etc.

This program creates a random-access file called JIM, and makes the 8th piece of data in JIM be "I LOVE LUCY", the 50th piece of data in JIM be "PLEASE WASH MY CAR", and the 6th piece of data in JIM be "DON'T PICK YOUR NOSE IN PUBLIC":

DEC using BASIC-PLUS or BASIC-PLUS-2

```
5 OPEN "JIM" AS FILE 1: DIM#1, X$(50)=30

10 X$(8)="I LOVE LUCY"
20 X$(50)="PLEASE WASH MY CAR"
30 X$(6)="DON'T PICK YOUR NOSE IN PUBLIC"

40 CLOSE 1
```

That line contains the number 50, because JIM's last item is the 50th. If you're using BASIC-PLUS-2, rewrite that line as follows:
5 DIM#1, X$(50)=30: OPEN "JIM" AS FILE 1, VIRTUAL

IBM PC

```
5 OPEN "JIM" AS 1: FIELD 1, 30 AS X$

10 LSET X$="I LOVE LUCY": PUT 1,8
20 LSET X$="PLEASE WASH MY CAR": PUT 1,50
30 LSET X$="DON'T PICK YOUR NOSE IN PUBLIC": PUT 1,6

40 CLOSE 1
```

CP/M, Radio Shack

```
5 OPEN "R",1,"JIM": FIELD 1, 30 AS X$

10 LSET X$="I LOVE LUCY": PUT 1,8
20 LSET X$="PLEASE WASH MY CAR": PUT 1,50
30 LSET X$="DON'T PICK YOUR NOSE IN PUBLIC": PUT 1,6

40 CLOSE 1
```

On the Color Computer, replace "R" by "D".

Commodore

```
5 OPEN 2,8,2,"JIM,L,"+CHR$(1+30)

10 PRINT#2, "P"CHR$(2)CHR$(8)CHR$(0): PRINT#2, "I LOVE LUCY"
20 PRINT#2, "P"CHR$(2)CHR$(50)CHR$(0): PRINT#2, "PLEASE WASH MY CAR"
30 PRINT#2, "P"CHR$(2)CHR$(6)CHR$(0): PRINT#2, "DON'T PICK YOUR NOSE IN PUBLIC"

40 CLOSE 2
```

Apple

```
5 D$=CHR$(4): ?D$"OPEN JIM,L30"

10 ?D$"WRITE JIM,R" 8: PRINT "I LOVE LUCY"
20 ?D$"WRITE JIM,R" 50: PRINT "PLEASE WASH MY CAR"
30 ?D$"WRITE JIM,R" 6: PRINT "DON'T PICK YOUR NOSE IN PUBLIC"

40 ?D$"CLOSE JIM"
```

In those programs, line 5 mentions the number 30, because the longest item in JIM has 30 characters.

This program makes the computer tell you what JIM's 8th item is:

DEC using BASIC-PLUS or BASIC-PLUS-2

```
5 OPEN "JIM" AS FILE 1: DIM#1, X$(50)=30
10 PRINT X$(8)
20 CLOSE 1
```

If you're using BASIC-PLUS-2, rewrite that line as follows: 5 DIM#1, X$(50)=30: OPEN "JIM" AS FILE 1, VIRTUAL

IBM PC

```
5 OPEN "JIM" AS 1: FIELD 1, 30 AS X$
10 GET 1,8: PRINT X$
20 CLOSE 1
```

CP/M, Radio Shack

```
5 OPEN "R",1,"JIM": FIELD 1, 30 AS X$
10 GET 1,8: PRINT X$
20 CLOSE 1
```

On the Color Computer, replace "R" by "D".

Commodore

```
5 OPEN 2,8,2,"JIM,L,"+CHR$(1+30)
10 PRINT#2, "P"CHR$(2)CHR$(8)CHR$(0): INPUT#2, X$: PRINT X$
20 CLOSE 2
```

Apple

```
5 D$=CHR$(4): ?D$"OPEN JIM,L30"
10 ?D$"READ JIM,R" 8: INPUT X$: PRINT X$
20 ?D$"CLOSE JIM"
```

Here are more details about random-access data files:

DEC using BASIC-PLUS or BASIC-PLUS-2

If JIM contains numbers instead of strings, the DIM part of line 5 should say just:

```
DIM#1, X(50)
```

If JIM contains strings *and* numbers, the DIM part of line 5 should say:

```
DIM#1, X$(50)=30, Y(50)
```

IBM PC, CP/M, Radio Shack

If you want JIM to contain *pairs* of strings, change the FIELD part of line 5 to this—

```
FIELD 1, 30 AS X$, 4 AS Y$
```

and say:

```
10 LSET X$="I LOVE LUCY": LSET Y$="WOW!": PUT 1,8
```

On IBM PC, CP/M, and Radio Shack computers, a random-access file must contain strings, not numbers; if you *want* to store numbers, you must convert them into strings.

CREATE A DATA BANK

CHRONOLOGICAL ORDER

The following program creates a data bank. In the data bank, you can store information about love, death, your friends, your enemies, your business, birthdays, bills, appointments, and whatever else bothers you. After storing information in the data bank, you can peek at the information, change it, expand on it, delete it, or do whatever else strikes your fancy.

The program consists of a main routine and 7 subroutines. It works on a Radio Shack model 3, but can be modified easily to work on other computers also.

For example, if you have an Apple, just delete line 10 (which the Apple can't understand), delete line 30 (which the Apple doesn't need), and rewrite the lines that contain the word ELSE (lines 60 and 5020).

The main routine

Increase the speed, by telling the computer that all variables are integers.

```
10 DEFINT A-Z
```

Allow 100 topics and corresponding data.

```
20 DIM T$(100), D$(100)
```

Reserve space in the RAM, to hold string variables.

```
30 CLEAR 0: CLEAR MEM-2000
```

At the beginning, the number of topics in the data bank is zero.

```
40 N=0
```

Ask the human for a topic.

```
50 PRINT: PRINT "WHAT TOPIC INTERESTS YOU?": PRINT "(IF YOU'RE NOT SURE, TYPE A
QUESTION MARK)": INPUT T$
```

Do what the human requested.

```
60 IF T$="?" THEN GOSUB 1000 ELSE GOSUB 2000
```

Go to another topic.

```
70 GO TO 50
```

Subroutine 1000: tell the human what topics are in the data bank

If the data bank is empty, say so.

```
1000 IF N=0 THEN PRINT "I DON'T KNOW ANY TOPICS YET.": PRINT "MY MIND IS STILL
BLANK.": PRINT "PLEASE TEACH ME A NEW TOPIC.": RETURN
```

If the data bank isn't empty, list the topics.

```
1010 PRINT "I KNOW ABOUT THESE TOPICS:": FOR I = 1 TO N: PRINT T$(I),: NEXT I:
PRINT: PRINT "PICK ONE OF THOSE TOPICS, OR TEACH ME A NEW ONE.": RETURN
```

Subroutine 2000: search through the data bank, to find the topic T$

If the topic is in the data bank, do subroutine 3000.

```
2000 FOR I = 1 TO N
2010     IF T$=T$(I) THEN GOSUB 3000: RETURN
2020 NEXT
```

If the topic isn't in the data bank, do subroutine 4000.

```
2030 GOSUB 4000
2040 RETURN
```

Subroutine 3000: the topic is in the data bank

Tell the human about the topic.
```
3000 PRINT "HERE'S WHAT I KNOW ABOUT "T$":": PRINT D$(I)
```

Ask the human whether to change that information.
```
3010 INPUT "DO YOU WANT TO CHANGE THAT INFORMATION";A$
```

If the human says YES or Y, do subroutine 5000, which changes the information.
```
3020 IF A$="YES" OR A$="Y" THEN GOSUB 5000
```

End subroutine 3000.
```
3030 RETURN
```

Subroutine 4000: the topic is not in the data bank

Say the topic is not in the data bank.
```
4000 PRINT "I DON'T KNOW ANYTHING ABOUT "T$"."
```

Request information about the topic.
```
4010 PRINT: PRINT "TELL ME ABOUT "T$: PRINT "(IF YOU DON'T WANT TO TELL ME, TYPE
THE LETTER X)": INPUT D$
```

If the human wishes, insert the topic into the data bank.
```
4020 IF D$<>"X" THEN GOSUB 6000
```

End subroutine 4000.
```
4030 RETURN
```

Subroutine 5000: change the information

Agree to change the information.
```
5000 PRINT "OKAY. I'VE ERASED THAT INFORMATION ABOUT "T$"."
```

Request new information.
```
5010 PRINT: PRINT "TYPE WHAT YOU WANT ME TO KNOW ABOUT "T$: PRINT "(IF YOU WANT
ME TO FORGET "T$", TYPE THE LETTER X)": INPUT D$
```

Change the information as the human requested.
```
5020 IF D$="X" THEN GOSUB 7000 ELSE D$(I)=D$
```

End subroutine 5000.
```
5030 RETURN
```

Subroutine 6000: insert the topic into the data bank

Increase the number of topics.
```
6000 N=N+1
```

Put the topic and data at the end of the data bank.
```
6010 T$(N)=T$: D$(N)=D$
```

End subroutine 6000.
```
6020 RETURN
```

Subroutine 7000: delete the topic from the data bank

Replace the topic by topic #N.
```
7000 T$(I)=T$(N): D$(I)=D$(N)
```

Decrease the number of topics.
```
7010 N=N-1
```

End subroutine 7000.
```
7020 RETURN
```

That program stores the topics in *chronological order*. In other words, if you feed it information about SUE and then information about CAROL, it will let T$(1) be SUE and let T$(2) be CAROL.

ALPHABETICAL ORDER

Instead of chronological order, you might prefer *alphabetical order*. For example, suppose you feed the computer information about SUE then CAROL then ZELDA then ALICE then JANE. Here's what the computer's memory would look like, in each kind of order:

Chronological order	Alphabetical order
SUE	ALICE
CAROL	CAROL
ZELDA	JANE
ALICE	SUE
JANE	ZELDA

Which is better— chronological order or alphabetical order? In chronological order, you can quickly *insert* a new name (just tack the name onto the bottom of the list), but to *find* a name in the list is slow (since the list looks disorganized). In alphabetical order, you can quickly *find* a name in the list (since the list is alphabetized), but to *insert* a new name in the list is slow (since the only way to insert a new name is to shove other names out of the way, to make room for it).

So which is better?

Chronological order is the *simplest* to program and is also the fastest for *inserting*. Alphabetical order is the fastest for *finding* information.

The program for alphabetical order resembles the program for chronological order. In fact, the main routine and subroutines 1000, 3000, 4000, and 5000 are *identical* to the ones for chronological order! The only differences lie in subroutines 2000, 6000, and 7000. Here are subroutines 2000, 6000, and 7000, rewritten for alphabetical order:

Subroutine 2000A: search through the data bank, to find the topic T$

```
Create L and H.
2000 L=0: H=N+1

Let I be the average of L and H.
2010 I=INT((L+H+1)/2)

If I=H, the topic isn't in the data bank, so do subroutine 4000.
2020 IF I=H THEN GOSUB 4000: RETURN

If the topic is found, do subroutine 3000.
2030 IF T$=T$(I) THEN GOSUB 3000: RETURN

Since the topic is not yet found, continue the search, by changing H or L and trying agai
2040 IF T$<T$(I) THEN H=I ELSE L=I
2050 GO TO 2010
```

Subroutine 6000A: insert the topic into the data bank

```
Increase the number of topics.
6000 N=N+1

Move other topics towards the end, to make room for the new topic.
6010 FOR J = N TO I+1 STEP -1: T$(J)=T$(J-1): D$(J)=D$(J-1): NEXT

Insert the new topic and its data.
6020 T$(I)=T$: D$(I)=D$

End subroutine 6000.
6030 RETURN
```

Subroutine 7000A: delete the topic from the data bank

Decrease the number of topics.
```
7000 N=N-1
```

Close the gap where the deleted topic had been.
```
7010 FOR J = I TO N: T$(J)=T$(J+1): D$(J)=D$(J+1): NEXT
```

End subroutine 7000.
```
7020 RETURN
```

Subroutine 2000A runs faster than the chronological subroutine 2000, because searching through an alphabetical list is faster than searching through a chronological list. (To search through the alphabetical list super-quickly, subroutine 2000A uses a trick called *binary search*.)

Unfortunately, subroutines 6000A (which inserts) and 7000A (which deletes) run more *slowly* than chronological subroutines 6000 and 7000; but if you want the high speed of subroutine 2000A, you must accept the slowness of subroutines 6000A and 7000A.

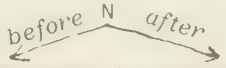
TREES

You've seen that chronological order is good for inserting and deleting but bad for searching, whereas alphabetical order is exactly the opposite: it's bad for inserting and deleting but good for searching. Which order do most businesses use, to store lots of data?

Most businesses use *neither* order! Instead, they use a *tree*, which is a structure that combines the best features of chronological order and alphabetical order. Like chronological order, a tree lets you insert and delete quickly. Like alphabetical order, a tree lets you search quickly also.

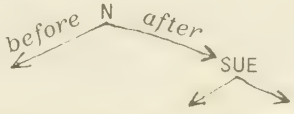
Do I therefore recommend that you use a tree? Not *always*. A tree is more complicated to program than chronological order or alphabetical order. If your boss insists that you finish writing a data-bank program *soon*, don't bother trying to create a tree. But if you can afford to spend the extra time and effort needed to create a tree, the speed at which the resulting program runs will delight you.

How do you create a tree? To understand the concept, begin by sketching a picture of a tree on paper. Since N is the middle letter of the alphabet, begin by writing the letter N, and put two arrows underneath it:

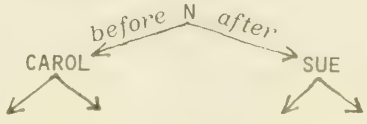


The left arrow is called the *before-arrow*; it will point to the names that come alphabetically before N. The right arrow is called the *after-arrow*; it will point to the names that come alphabetically after N.

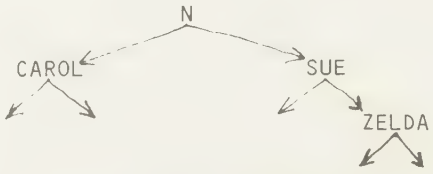
For example, suppose your first topic is SUE. Since SUE comes alphabetically after N, put SUE at the tip of N's after-arrow:



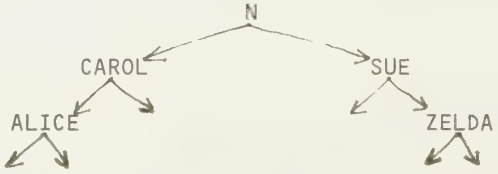
Suppose your next topic is CAROL. Since CAROL comes alphabetically before N, put CAROL at the tip of N's before-arrow:



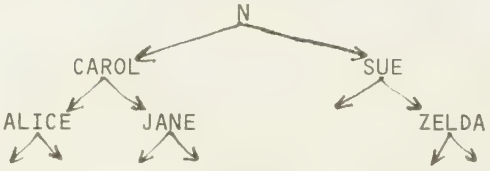
Suppose your next topic is ZELDA. Since ZELDA comes after N, we'd like to put ZELDA at the tip of N's after-arrow; but SUE's already stolen that position. So compare ZELDA against SUE. Since ZELDA comes after SUE, put ZELDA at the tip of SUE's after-arrow:



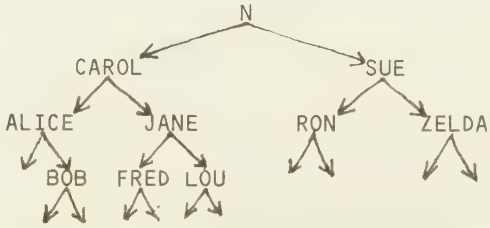
Suppose your next topic is ALICE. Since ALICE comes before N, look at the tip of N's before-arrow. Since CAROL's stolen that position, compare ALICE against CAROL; since ALICE comes before CAROL, put ALICE at the tip of CAROL's before-arrow:



Suppose your next topic is JANE. Since JANE comes before N, look at the tip of N's before-arrow. Since CAROL's stolen that position, compare JANE against CAROL; since JANE comes after CAROL, put JANE at the tip of CAROL's after-arrow:



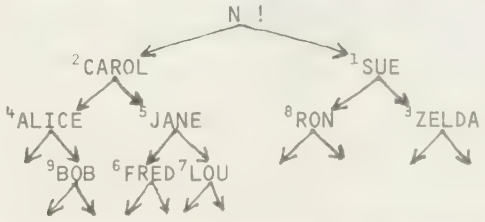
If the next few topics are FRED, then LOU, then RON, then BOB, the tree looks like this:



Again, look at the arrows that point down from N. N's before-arrow points to the group of names that come alphabetically before N (such as CAROL, ALICE, JANE, BOB, FRED, and LOU); N's after-arrow points to the group of names that come alphabetically after N (such as SUE, RON, and ZELDA). Similarly, CAROL's before-arrow points to the group of names that come alphabetically before CAROL (such as ALICE and BOB); CAROL's after-arrow points to a group of names that come alphabetically after CAROL (such as JANE, FRED, and LOU).

Programmers treat the tree as if it were a "family tree". CAROL is called the *parent* of ALICE and JANE, who are therefore called CAROL's *children*. CAROL is called the *ancestor* of ALICE, JANE, BOB, FRED, and LOU, who are therefore called CAROL's *descendants*. The arrows are called *pointers*.

To make the tree more useful, begin with "N !" instead of "N" (so that you choose "N" as a topic later), and number the topics in the order they appeared: since SUE was the first topic, put "1" in front of SUE; since CAROL was the second topic, put "2" in front of CAROL; since ZELDA was the third topic, put "3" in front of ZELDA, like this:



To describe the tree to the computer, store this table in the computer's memory:

Topic	Where the before-arrow points	Where the after-arrow points
0N !	2	1
1SUE	8	3
2CAROL	4	5
3ZELDA	0	0
4ALICE	0	9
5JANE	6	7
6FRED	0	0
7LOU	0	0
8RON	0	0
9BOB	0	0

That table *represents* the tree. That table is called the tree's *representation*.

The table's left column is in chronological order, but the other columns give information about alphabetizing. So a tree combines chronological order with alphabetical order: it combines the advantages of both. Adding a new topic to the tree is quick and easy (as in chronological order): just add the name to the bottom of the list, and adjust a few arrows. Using the tree to search for a topic is quick and easy (as in alphabetical order): just follow the arrows.

The program that creates and manipulates the tree resembles the program for chronological order. In fact, subroutines 1000, 3000, 4000, and 5000 are identical to the ones for chronological order! The only differences lie in the main routine and subroutines 2000, 6000, and 7000 and a new subroutine that you must add (called "subroutine 8000"). Here are those new routines:

The main routine T

Increase the speed, by telling the computer that all variables are integers.
```
10 DEFINT A-Z
```

Allow 100 topics and their data and pointers.
```
20 DIM T$(100), D$(100), P(100,2)
```

Reserve space in the RAM, to hold string variables.
```
30 CLEAR 0: CLEAR MEM-200
```

Topic #0 is "N !".
```
40 T$(0)="N !": P(0,1)=0: P(0,2)=0
```

At the beginning, the number of topics in the data file is considered to be 0.
```
50 N=0
```

Ask the human for a topic.
```
60 PRINT: PRINT "WHAT TOPIC INTERESTS YOU?": PRINT "(IF YOU'RE NOT SURE, TYPE A
QUESTION MARK)": INPUT T$
```

Do what the human requested.
```
70 IF T$="?" THEN GOSUB 1000 ELSE GOSUB 2000
```

Go to another topic.
```
80 GO TO 60
```

Subroutine 2000T: search through the data bank, to find the topic T$

Starting at position 0, try to find T$.
```
2000 I=0: GOSUB 8000
```

If T$ is missing from the data bank, do subroutine 4000; otherwise do subroutine 3000.
```
2010 IF I=0 THEN GOSUB 4000 ELSE GOSUB 3000
```

End subroutine 2000.
```
2020 RETURN
```

Subroutine 6000T: insert the topic into the file

Increase the number of topics.
```
6000 N=N+1
```

Put that topic and its data at the end of the data bank.
```
6010 T$(N)=T$: D$(N)=D$: P(N,1)=0: P(N,2)=0
```

Let topic I1 point to the new topic (which is topic N).
```
6020 P(I1,J)=N
```

End subroutine 6000.
```
6030 RETURN
```

Subroutine 7000T: delete the topic from the data bank

Let topic I1 (which had been pointing to the vanishing topic) point to the vanishing topic's left child instead.

```
7000 P(I1,J)=P(I,1)
```

Let A be the number of the vanishing topic, and let B be the number of the vanishing topic's right child.

```
7010 A=I: B=P(I,2)
```

If the vanishing topic has a right child, make something point to that child.

```
7020 IF B>0 THEN T$=T$(B): I=I1: GOSUB 8000: P(I1,J)=B
```

If the vanishing topic isn't topic N, move topic N to the gap left by the vanishing topic.

```
7030 IF A<N THEN T$=T$(N): I=0: GOSUB 8000: P(I1,J)=A: T$(A)=T$: D$(A)=D$(N):
P(A,1)=P(N,1): P(A,2)=P(N,2)
```

Decrease the number of topics.

```
7040 N=N-1
```

End subroutine 7000.

```
7050 RETURN
```

Subroutine 8000T: hunt for T$ in the data bank, as follows....*Start hunting at position I. When the search is done, let I = the position of T$; let I1 = the position of the topic that points to T$; let J = 1 if the arrow pointing to T$ points to the left, but let J = 2 if that arrow points to the right. If T$ isn't in the data bank, let I = 0.*

```
8000 IF T$=T$(I) THEN RETURN
8010 IF T$<T$(I) THEN J=1 ELSE J=2
8020 I1=I: I=P(I,J)
8030 IF I>0 THEN GO TO 8000
8040 RETURN
```

The data bank programs we've written put data into the RAM. Unfortunately, when you turn off the power, the RAM forgets all the data. Let's create a better program, that puts the data bank onto a disk instead of into RAM.

The following program puts the data bank onto a disk, by using a tree. It's written for the Radio Shack model 1 or model 3. If you have an IBM Personal Computer or a CP/M computer, you can run the program with only minor changes. (To run the program on an Apple, put a Z-80 Softcard into your Apple, so that your Apple can handle CP/M.)

The program uses three features of advanced BASIC:

Feature	Meaning
LINE INPUT	INPUT a whole line, even if the line contains commas.
MKI$	Turn an integer into a string.
CVI	Turn a string into an integer.

It also uses these instructions for handling random-access files:

```
OPEN...CLOSE
PUT...GET
FIELD...LSET
LOF
```

Here's the program:

The main routine D

Prepare the variables.

```
10 DEFINT A-Z: DIM PF$(2): CLEAR 1000: Z$=MKI$(0)   For CP/M or IBM, omit "CLEAR 1000".
```

Open the file and its fields.

```
20 OPEN "R",1,"MASTER/DAT"   For CP/M or IBM, change "MASTER/DAT" to "MASTER.DAT".
30 FIELD 1, 60 AS TF$, 191 AS DF$, 2 AS PF$(1), 2 AS PF$(2)   For CP/M, change 191 to 6
```

If the file's still empty, let topic #1 be the header and topic #2 be "N !".

```
40 IF LOF(1)=0 THEN LSET PF$(1)=MKI$(2): PUT 1: LSET TF$="N !": LSET PF$(1)=Z$:
LSET PF$(2)=Z$: PUT 1
```

Get N from the file.

```
50 GET 1,1: N=CVI(PF$(1))
```

Ask the human for a topic.

```
60 PRINT: PRINT "WHAT TOPIC INTERESTS YOU?": PRINT "(IF YOU'RE NOT SURE, TYPE A
QUESTION MARK)": PRINT "(IF YOU WANT TO END, TYPE THE LETTER X)": LINE INPUT T$
```

Handle unusual input.

```
70 IF T$="?" THEN GOSUB 1000: GO TO 60
80 IF T$="X" THEN CLOSE 1: END
```

Make TS$ resemble T$; but make TS$ be exactly 60 characters long, by adding blank spaces at the end.

```
90 LSET TF$=T$: TS$=TF$
```

Find the topic.

```
100 GOSUB 2000
```

Go to another topic.

```
110 GO TO 60
```

Subroutine 1000D: tell the human what topics are in the file

If the file is empty (aside from the header and "N !"), say so.

```
1000 IF N=2 THEN PRINT "I DON'T KNOW ANY TOPICS YET.": PRINT "MY MIND IS STILL
BLANK.": PRINT "PLEASE TEACH ME A NEW TOPIC.": RETURN
```

If the file isn't empty, list the topics.

```
1010 PRINT "I KNOW ABOUT THESE TOPICS:": FOR I = 3 TO N: GET 1,I: PRINT TF$: NEXT:
PRINT "PICK ONE OF THOSE TOPICS, OR TEACH ME A NEW ONE.": RETURN
```

Subroutine 2000D: search through the file, to find the topic TS$

Starting at position 2, try to find TS$.

```
2000 I=2: GOSUB 8000
```

If TS$ is missing from the file, do subroutine 4000; otherwise do subroutine 3000.

```
2010 IF I=0 THEN GOSUB 4000 ELSE GOSUB 3000
```

End subroutine 2000.

```
2020 RETURN
```

Subroutine 3000D: the topic is in the file

Tell the human about the topic.

```
3000 PRINT "HERE'S WHAT I KNOW ABOUT "T$":": PRINT DF$
```

Ask the human whether to change that information.

```
3010 INPUT "DO YOU WANT TO CHANGE THAT INFORMATION";A$
```

If the human says YES or Y, do subroutine 5000, which changes the information.

```
3020 IF A$="YES" OR A$="Y" THEN GOSUB 5000
```

End subroutine 3000.

```
3030 RETURN
```

Subroutine 4000D: the topic is not in the file

Say that the topic isn't in the file.

```
4000 PRINT "I DON'T KNOW ANYTHING ABOUT "T$"."
```

Request information about the topic.

```
4010 PRINT: PRINT "TELL ME ABOUT "T$: PRINT "(IF YOU DON'T WANT TO TELL ME, TYPE
THE LETTER X)": LINE INPUT D$
```

If the human wishes, insert that topic into the file.

```
4020 IF D$<>"X" THEN GOSUB 6000
```

End subroutine 4000.

```
4030 RETURN
```

Subroutine 5000D: change the information

Agree to change the information.

```
5000 PRINT "OKAY. I'VE ERASED THAT INFORMATION ABOUT "T$"."
```

Request new information.

```
5010 PRINT: PRINT "TYPE WHAT YOU WANT ME TO KNOW ABOUT "T$: PRINT "(IF YOU WANT
ME TO FORGET "T$", TYPE THE LETTER X)": LINE INPUT D$
```

Change the information as the human requested.

```
5020 IF D$="X" THEN GOSUB 7000 ELSE LSET DF$=D$: PUT 1,I
```

End subroutine 5000.

```
5030 RETURN
```

Subroutine 6000D: insert the topic into the file

Let topic I1 point to the new topic, which is topic N+1.

```
6000 LSET PF$(J)=MKI$(N+1): PUT 1,I1
6010 LSET TF$=TS$: LSET DF$=D$: LSET PF$(1)=Z$: LSET PF$(2)=Z$: PUT 1,N+1
```

Increase the number of topics.

```
6020 N=N+1: LSET PF$(1)=MKI$(N): PUT 1,1
```

End subroutine 6000.

```
6030 RETURN
```

Subroutine 7000D: delete the topic from the file

Let A be the number of the vanishing topic, and let P1 and P2 be the topic's pointers.

```
7000 A=I: P1=CVI(PF$(1)): P2=CVI(PF$(2))
```

Let topic I1 (which had been pointing to the vanishing topic) point to the vanishing topic's left child instead.

```
7010 GET 1,I1: LSET PF$(J)=MKI$(P1): PUT 1,I1
```

If the vanishing topic has a right child, make something point to that child.

```
7020 IF P2>0 THEN GET 1,P2: TS$=TF$: I=I1: GOSUB 8000: LSET PF$(J)=MKI$(P2): PUT 1,I1
```

If the vanishing topic isn't topic N, move topic N to the gap left by the vanishing topic.

```
7030 IF A<N THEN GET 1,N: PUT 1,A: TS$=TF$: I=2: GOSUB 8000: GET 1,I1:
LSET PF$(J)=MKI$(A): PUT 1,I1
```

Decrease the number of topics.

```
7040 N=N-1: LSET PF$(1)=MKI$(N): PUT 1,1
```

End subroutine 7000.

```
7050 RETURN
```

Subroutine 8000D: hunt for TS$ in the file, as follows....*Start hunting at position I. When the search is done, let I = the position of TS$; let I1 = the position of the topic that points to TS$; let J = 1 if the arrow pointing to TS$ points to the left, but let J = 2 if that arrow points to the right. If TS$ is not in the file, let I = 0.*

```
8000 GET 1,I
8010 IF TS$=TF$ THEN RETURN
8020 IF TS$<TF$ THEN J=1 ELSE J=2
8030 I1=I: I=CVI(PF$(J))
8040 IF I>0 THEN GO TO 8000
8050 RETURN
```

JARGON

Suppose you walk into an old-fashioned office that doesn't have a computer. In that office, you'll find a filing cabinet that contains several drawers. One drawer's labeled "customers"; another drawer's labeled "employees"; another drawer's labeled "suppliers". Each drawer contains many "index cards", alphabetized.

For example, the drawer labeled customers contains a card about each customer: the first card might be labeled "ADAMS, JOAN"; the last card might be labeled "ZABRONSI, JILL". The first card contains all known information about Joan Adams: it contains her name, address, phone number, a list of everything she's purchased from the office, how much she paid, how much she still owes, and other personal information about her. That card is called her *record*. Each item of information on that card is called a *field*. If the card is a pre-printed form, it allows a certain amount of space for each item: for example, it might allow only 30 characters for the person's name. The number of characters allowed for a field is called the field's *width*. In that example, the width of the NAME field is 30 characters.

Each drawer is called a *file*. For example, the drawer that contains information about customers is called the *customer file*; another drawer is the *employee file*; another drawer is the *supplier file*.

The entire filing cabinet— which contains all the information about your company — is called the *data base*.

A computerized system that handles an entire data base well is called a *data base management system (DBMS)*. To be given the honor of being called a DBMS, the computerized system must be so well organized that *each data item appears only once*.

For example, if Joan Adams is one of the customers, her name should appear in the data base only once. That way, if she's so old-fashioned that she gets married and changes her name, only *one* place in the data base has to be changed.

To dig deeper into the problem of changing Joan's name, look at the file of employees. Suppose it includes the names of all the employees in alphabetical order. Suppose one of the employees is Joe Schwartz, who sold a hair dryer to Joan Adams. In a badly constructed data base, Joe Schwartz's card (which is called "SCHWARTZ, JOE") contains a note that he sold a hair dryer to Joan Adams; when Joan Adams changes her name, I'll betcha that either the clerk or the computer will forget to change the note on Joe Schwartz's card!

To avoid that problem, Joe Schwartz's card should *not* contain the name "Joan Adams". Instead, it should contain the *number* of Joan Adams' card. For example, if Joan Adams is customer #2034, Joe's card should say that he sold a hair dryer to "customer #2034". Even if Joan changes her name, the statement that Joe Schwartz sold a hair dryer to customer #2034 is still accurate.

Joe Schwartz's card says he sold a hair dryer to "customer #2034". On his card, the notation "see customer #2034" means: for details about who Joe Schwartz sold the hair dryer to, see card #2034 in the customer file. So Joe's card refers you to customer card #2034. The notation on Joe's card that says "see customer card #2034" is called a *cross-reference* (since it refers you to card #2034); it's also called a *pointer* (since it points you to card #2034). Every DBMS needs such cross-references (pointers), to interrelate the files in the data base.

Now suppose you want to find out who Joe Schwartz sold the hair dryer to. If you look on Joe's card, you won't find Joan Adams' new name or even her old name; all you'll find is "customer #2034". How do you find out who customer #2034 is? That's why you need a computer: you tell the computer to hunt through the customer file, to find the name of customer #2034.

CHAPTER 4

FORTRAN

WHAT'S FORTRAN?

Earlier in *The Secret Guide to Computers*, I explained BASIC. Now you'll learn a different computer language, called *FORTRAN*. Most maxicomputers and minicomputers understand both BASIC and FORTRAN. Some ideas are easier to express in BASIC; others are easier in FORTRAN. Most scientists and engineers on large computers use FORTRAN, not BASIC.

IBM invented the first version of FORTRAN in 1957. Then came improvements, called *FORTRAN II*, *FORTRAN III*, and *FORTRAN IV*. The newest version is called *FORTRAN 77*, because it was invented in 1977. Today, some computers use FORTRAN 77; others still use FORTRAN IV, or a slightly souped-up version of it (called *FORTRAN IV-EXTENDED* or *FORTRAN V* or *WATFOR* or *WATFIV* or *FORTRAN 10*). But have no fear: the features of FORTRAN discussed in this chapter work on practically all computers.

SIMPLE PROGRAMS

Here's a FORTRAN program:

```
      PRINT 10
10    FORMAT (1X,'CHIPMUNKS ARE CHUBBY')
      PRINT 20
20    FORMAT (1X,'GOLDFISH GIGGLE')
      PRINT 10
      END
```

The top line says to print what's in line 10, so the computer will print CHIPMUNKS ARE CHUBBY. The next line says to print what's in line 20, so the computer will print GOLDFISH GIGGLE. The next line says to print what's in line 10, so the computer will print CHIPMUNKS ARE CHUBBY again. Altogether, the program makes the computer print:

```
CHIPMUNKS ARE CHUBBY
GOLDFISH GIGGLE
CHIPMUNKS ARE CHUBBY
```

Notice:

Each line of the program is indented 6 spaces, so it begins in the 7th position.
Each FORMAT begins with 1X.
Each string is enclosed in apostrophes.

You must number every line that's referred to. For example, you must number the FORMAT lines, since the PRINT lines refer to them. You don't have to number the PRINT lines.

The bottom line of every FORTRAN program must be END.

That program works on most computers. On PDP computers, say TYPE instead of PRINT. On CDC computers using TS FORTRAN, replace each apostrophe by an asterisk. On IBM computers, put STOP above END, so the bottom two lines of your program are:

```
      STOP
      END
```

Ask the people in your computer center how to feed a FORTRAN program into the computer. On some computers, you must put an edit number in front of each line:

```
00100          PRINT 10
00110    10    FORMAT (1X,'CHIPMUNKS ARE CHUBBY')
00120          PRINT 20
00130    20    FORMAT (1X,'GOLDFISH GIGGLE')
00140          PRINT 10
00150          END
```

Some computers don't require you to indent each line. To indent quickly on PDP computers, hold down the CONTROL key, while you type the letter I.

Each line of your program must be short: no more than 72 characters, including the spaces at the beginning of the line.

CARRIAGE CONTROLS

The 1X at the beginning of each FORMAT is called the *carriage control*. It means: print the format normally. For weirder printing, replace the 1X by '0' or '1' or '+':

```
      PRINT 10
10    FORMAT (1X,'NIFTY')
      PRINT 20
20    FORMAT ('0','SAL')
      END
```

In line 10, the 1X makes the computer print NIFTY normally. In line 20, the zero-in-apostrophes makes the computer leave a blank line, then print SAL. The computer will print:

```
NIFTY

SAL
```

Suppose you change the carriage control to '1':

```
      PRINT 10
10    FORMAT (1X,'NIFTY')
      PRINT 20
20    FORMAT ('1','SAL')
      END
```

If your terminal uses paper, the '1' makes the computer print SAL on a new page. If your terminal uses a screen instead of paper, the '1' makes the computer erase the screen before printing SAL.

Suppose you change the carriage control to '+':

```
      PRINT 10
10    FORMAT (1X,'NIFTY')
      PRINT 20
20    FORMAT ('+','SAL')
      END
```

If your terminal uses paper, the '+' makes the computer print SAL in the same place as NIFTY, like this:

```
SALTY
```

If your terminal uses a screen instead, the computer will print NIFTY, but then the NIF will suddenly disappear, and you'll see SALTY.

To print the symbol θ, print 0 in the same place as -. To print the symbol ≠, print = in the same place as /. To print θ≠A, print 0=A in the same place as -/. If your terminal uses paper, this program prints θ≠A:

```
      PRINT 10
10    FORMAT (1X,'0=A')
      PRINT 20
20    FORMAT ('+','-/')
      END
```

If you're using a Hazeltine terminal or CDC TS FORTRAN, the carriage controls won't work.

Examine this program:

```
      PRINT 10
10    FORMAT (1X,'JOHN','NY')
      END
```

The FORMAT consists of three *fields*: the first is the carriage control 1X, the second is 'JOHN', and the third is 'NY'. The computer will print JOHN and NY on the same line:

```
JOHNNY
```

Another example:

```
      PRINT 10
10    FORMAT (1X,'EAT A',8X,'MEATBALL')
      END
```

The computer will print EAT A, then 8 blank spaces, then MEATBALL:

```
EAT A        MEATBALL
```

Here's another program for meatball lovers:

```
      PRINT 10
10    FORMAT (1X,'EAT A',1X,'MEATBALL')
```

The computer will print EAT A, then 1 blank space, then MEATBALL:

```
EAT A MEATBALL
```

If you say X instead of 1X, the computer will gripe.

This program quotes Julius Caesar:

```
      PRINT 10
10    FORMAT (1X,'I CAME')
      PRINT 20
20    FORMAT (1X,'I SAW')
      PRINT 30
30    FORMAT (1X,'I CONQUERED')
      END
```

The computer will print:

```
I CAME
I SAW
I CONQUERED
```

This program does the same thing:

```
      PRINT 10
10    FORMAT (1X,'I CAME'/1X,'I SAW'/1X,'I CONQUERED')
      END
```

Line 10 consists of 3 *records*: the first is 1X,'I CAME'; the second is 1X,'I SAW'; the third is 1X,'I CONQUERED'. Each record begins with a carriage control; the records are separated by slashes. The computer will print each record on a separate line.

GO TO

You can say GO TO:

```
10    PRINT 20
20    FORMAT (1X,'CAT')
      PRINT 30
30    FORMAT (1X,'DOG')
      GO TO 10
      END
```

The top line says to print what's in line 20, so the computer will print:

```
CAT
```

The next line says to print what's in line 30, so the computer will print:

```
DOG
```

The next line makes the computer go back to line 10. The computer will print CAT again, then DOG again, then jump back to line 10 again....The computer will try to print the words CAT and DOG again and again, forever.

STOP

The computer understands the word STOP:

```
      PRINT 10
10    FORMAT (1X,'BUBBLE GUM')
      PRINT 20
20    FORMAT (1X,'SHAKESPEARE')
      STOP
      PRINT 30
30    FORMAT (1X,'DADA')
      END
```

The top line says to print what's in line 10, so the computer will print BUBBLE GUM. The next line says to print what's in line 20, so the computer will print SHAKESPEARE. The next line says STOP, so the computer will stop. It will never print DADA.

MATH

NUMBERS

FORTRAN distinguishes between *integers* and *real numbers*. Here's how FORTRAN defines them....

An *integer* contains no decimal point and no exponent.

Integer	Not an integer	Comment
-27	-27.0	An integer contains no decimal point.
50000	5E4	An integer contains no exponent.

A *real number* contains either a decimal point or the letter E.

Real number	Not a real number
-27.0	-27
2.35E8	235000000
5E4	50000

The largest permissible integer is different from the largest permissible real:

Computer	Largest integer	Largest real	Tiniest real
PDP-11 using FORTRAN IV	32767	1.7E38	2.9E-39
PDP-11 using FORTRAN IV-PLUS	2147483647	1.7E38	2.9E-39
PDP-10 or Honeywell	34359738367	1.7E38	2.9E-39
IBM	2147483647	7.2E75	5.4E-79
CDC	281474976710655	2.5E322	3.1E-294

In BASIC, X can stand for a number, such as 3.7. The same is true in FORTRAN. A variable can be a letter (such as X) or a letter-followed-by-a-combination-of-letters-and-digits (such as FUN4U2). The variable must be short: no more than 6 characters. AVERAGE is too long: say AVERAG instead. If the variable begins with I, J, K, L, M, or N, it stands for an integer. If it begins with some other letter, it stands for a real number.

Example:

```
      JUNKY=-47
      PRINT 10, JUNKY
10    FORMAT (1X,I3)
      END
```

Since JUNKY begins with J, it stands for an integer. The first line says JUNKY stands for the integer -47. The second line says to print JUNKY, using line 10. Line 10 explains how to print JUNKY. The I3 means: print it as an Integer having 3 characters. The computer will print:

```
-47
```

If you change the I3 to I4, the computer will print JUNKY as an Integer having 4 characters. To print a total of 4 characters, the computer will print a blank space in front of -47, like this:

```
 -47
```

If you change to I5, the computer will print JUNKY as an Integer having 5 characters, by printing two blank spaces in front of -47:

```
  -47
```

If you change to I2, the computer will try to print JUNKY as an integer having 2 characters. But it's impossible to express -47 by using only 2 characters. The computer will obey the format and print 2 characters, but will make them asterisks:

```
**
```

Another example:

```
      NUM=31.9
      PRINT 10, NUM
10    FORMAT (1X,I4)
      END
```

Since NUM begins with N, it stands for an integer. The program's top line tries to make NUM stand for 31.9; but that's impossible, since 31.9 isn't an integer. The computer will omit the .9 and make NUM stand for the integer 31. The computer will print 31, using an I4 format:

```
 31
```

Example:

```
      JOE=-5.8
      PRINT 10, JOE
10    FORMAT (1X,I4)
      END
```

The computer will set JOE equal to the integer -5 and print it:

```
-5
```

Example:

```
      JAIL=74
      KRIMNL=829
      PRINT 10, JAIL,KRIMNL
10    FORMAT (1X,I2,I3)
      END
```

Since JAIL begins with J, and KRIMNL begins with K, they're both integers. The computer will print JAIL and KRIMNL, using the format in line 10. The format says to print a 2-character integer, then a 3-character integer. The computer will print:

```
74829
```

If you change the format to (1X,I2,4X,I3), the computer will print a 2-character integer, then 4 blanks, then a 3-character integer:

```
74    829
```

If you change the format to—

```
10    FORMAT (1X,'JAIL NUMBER',1X,I2,1X,'CONTAINS CRIMINAL',1X,I3)
```

the computer will print JAIL NUMBER, then a blank, then a 2-character integer, then a blank, then CONTAINS CRIMINAL, then a blank, then a 3-character integer:

```
JAIL NUMBER 74 CONTAINS CRIMINAL 829
```

Example:

```
      J=43
      K=75
      L=96
      M=81
      N=24
      PRINT 10, J,K,L,M,N
10    FORMAT (1X,I2,I2,I2,I2,I2)
      END
```

The computer will print 43, then 75, then 96, then 81, then 24:

```
4375968124
```

You can write that format more briefly:

```
10    FORMAT (1X,5I2)
```

If you change the format to (1X,5I3), the computer will print each integer as 3 characters— a blank followed by two digits:

```
 43 75 96 81 24
```

If you change the format to (1X,I3), the computer will print only one integer per line:

```
 43
 75
 96
 81
 24
```

If you change the format to (1X,2I3), the computer will print 2 integers per line:

```
 43 75
 96 81
 24
```

If you change the format to (1X,'GOSH',I3,1X,'SUPERB',I3,1X,'JEEPERS'), the computer will print 2 integers per line:

```
GOSH 43 SUPERB 75 JEEPERS
GOSH 96 SUPERB 81 JEEPERS
GOSH 24 SUPERB
```

To be safe, use I14 format for integers. On most computers, I14 handles even the largest integers, and prints blank spaces between them.

Integers are also called *fixed-point* numbers. Real numbers are called *floating-point* numbers.

The I format is only for integers. For real numbers, use F or G. The I format is easy to understand:

```
      RADIUS=-586.39
      PRINT 10, RADIUS
10    FORMAT (1X,F7.2)
      END
```

Since RADIUS doesn't begin with I, J, K, L, M, or N, it stands for a real number. The first line says RADIUS stands for the real number -586.39. The second line says to print RADIUS, using the format in line 10. The F7.2 means: print it as a floating-point number having 7 characters, 2 of them after the decimal point. The computer will print:

```
-586.39
```

If you change the F7.2 to a different format, the following chart shows what happens; in the chart, each ▮ represents a blank space:

Format	What the computer will print	Comment
F8.2	▮-586.39	To print 8 characters instead of 7, it prints a blank space at the beginning.
F8.3	-586.390	To print 3 characters after the decimal point instead of 2, it prints a zero at the end.
F8.1	▮▮-586.4	To print 1 character after the decimal point instead of 2, it rounds the .39 to .4.
F8.4	********	To print 4 characters after the decimal point, the computer would have to print -586.3900; since that requires more than 8 characters, the computer complains by printing asterisks.

To print a real number, the safest format is G14.6. The G14.6 handles all real numbers— even the largest— so it is more flexible than F. The G14.6 prints 14 characters altogether, 6 of which are significant digits. Here are examples of numbers printed in G14.6 format:

```
▮-0.283941E-29
▮▮0.293027▮▮▮▮
▮▮▮5.34523▮▮▮▮
▮▮▮39.4539▮▮▮▮
▮▮▮47802.3▮▮▮▮
▮▮▮986327.▮▮▮▮
▮▮0.288341E+24
```

Example:

```
      PRUNES=17
      PRINT 10, PRUNES
10    FORMAT (1X,G14.6)
      END
```

Since PRUNES doesn't begin with I, J, K, L, M, or N, it stands for a real number. When the computer encounters the first line of the program, it will set PRUNES equal to the real number 17.0. It will print:

```
 17.0000
```

The program will run faster if you change the top line to this:

```
      PRUNES=17.0
```

For addition, subtraction, multiplication, and division, FORTRAN uses the same symbols as BASIC.

```
      N=2*(3+1)
      S=7.3+2.1
      PRINT 10, N,S
10    FORMAT (1X,I14,G14.6)
      END
```

Since N is 8, and S is 9.4, the computer will print:

```
             8   9.40000
```

For exponents, FORTRAN uses a double star:

```
      J=7**2
      P=.5**3
      PRINT 10, J,P
10    FORMAT (1X,I14,G14.6)
      END
```

Since J is 7^2 (which is 49), and P is $.5^3$ (which is .125), the computer will print:

```
            49   0.125000
```

For negative exponents, you need parentheses. You must say 6.1**(-2), not 6.1**-2.

When you combine integers, the answer is an integer:

2+3 is 5
8-8 is 0
2*4 is 8
399/100 is 3 (not 3.99)
11/4 is 2 (not 2.75)
3/4 is 0 (not 0.75)
10**(-2) is 0 (not 0.01)

When you combine real numbers, the answer is real:

4.1+2.9 is 7.0 (not 7)
8.0-8.0 is 0.0 (not 0)
399.0/100.0 is 3.99
11.0/4.0 is 2.75
3.0/4.0 is .75
10.0**(-2.0) is .01

When you combine an integer with a real number, the answer is real:

3+2.0 is 5.0
399/100.0 is 3.99
11/4.0 is 2.75
3/4.0 is .75
10.0**(-2) is .01

Compare these:

7/10*10 is 0 (because 7/10 is 0)
7/10*10.0 is 0.0 (because 0*10.0 is 0.0)
7/10.0*10 is 7.0 (because 7/10.0 is .7)

Example:

```
      JERK=20.9+30.9
      PRINT 10, JERK
10    FORMAT (1X,I14)
      END
```

Since JERK begins with J, it stands for an integer. Since 20.9+30.9 is 51.8, JERK stands for the integer 51. The computer will print:

```
            51
```

Another example:

```
      APPLE=37/10
      PRINT 10, APPLE
10    FORMAT (1X,G14.6)
      END
```

Since APPLE begins with A, it stands for a real number. Since 37/10 is 3, APPLE stands for the real number 3.0. The computer will print:

```
 3.00000
```

If you commit one of these crimes, the computer will work slowly....

little crime: use a real number
medium crime: mix integers with reals
big crime: use a real exponent

For example, 2.0+2.0 is slower than 2+2, because 2.0+2.0 is a little crime. 2.1+7 is slower than 2.1+7.0, because 2.1+7 is a medium crime, whereas 2.1+7.0 is only a little crime. 5.1**2.0 is slower than 5.1**2, because 5.1**2.0 is a big crime, whereas 5.1**2 is only a medium crime. X=0 is slower than X=0.0, because X=0 is a medium crime, wnereas X=0.0 is only a little crime.

5**3.1 is a gigantic crime, since it is a medium crime and a big crime simultaneously. Because the crime is so gigantic, some computers refuse to do it. Say 5.0**3.1 instead.

ADVICE ABOUT VARIABLES

FORTRAN, like BASIC, distinguishes variables, constants, and expressions:

X is a variable
2.7 is not a variable; it's a *numeric constant*
'LOVE' is not a variable; it's a *string constant*
X+Y is not a variable; it's an *expression*

In a PRINT statement, some computers allow only variables....

allowed: PRINT 10, X
not allowed: PRINT 10, 2.7 *instead, say X=2.7 and PRINT 10, X*
not allowed: PRINT 10, 'LOVE' *instead, say PRINT 10 and 10 FORMAT (1X, 'LOVE')*
not allowed: PRINT 10, X+Y *instead, say Z=X+Y and PRINT 10, Z*

Other computers are more generous and allow anything. Find out about yours.

To help other humans understand your program, use long variable names throughout your program. Say RADIUS, not R; say AREA, not A; say VOLUME, not V; say SUM, not S; say TOTAL, not T. Because FORTRAN's variables are restricted to six characters, you might have to omit the last few syllables (*revolutions* becomes REVOLU) or the last few vowels (RVLTNS).

If you want a variable to be real, but its English name begins with I, J, K, L, M, or N, begin its FORTRAN name with an A (*mass* becomes AMAS; *length* becomes ALENGT or ALNGTH). If you want a variable to be an integer, but its English name doesn't begin with I, J, K, L, M, or N, begin its FORTRAN name with an I (*population* become IPOPUL) or misspell it (*count* becomes KOUNT) or choose a synonym (instead of *position*, say *location*, which is LOCATN).

READ

The computer can read.

```
      PRINT 10
10    FORMAT (1X,'TYPE SOME DIGITS')
      READ 20, N
20    FORMAT (I4)
      PRINT 30, N
30    FORMAT (1X,I4)
      END
```

When you run the program, the first pair of lines make the computer print:

```
TYPE SOME DIGITS
```

The word READ makes the computer wait for you to type something; it's like the BASIC word INPUT. The computer will wait for you to type the value of N, but line 20's FORMAT makes the computer read just the first 4 characters. For example, suppose you type:

```
-75198622
```

The computer will read just the first 4 characters, which are -751; it will ignore the 98622; so N will be -751. Line 30's FORMAT makes the computer print:

```
-751
```

Altogether, the run looks like this:

```
The computer says→→→→→TYPE SOME DIGITS
You say→→→→→→→→→→→→→→-75198622
The computer replies→→→-751
```

Line 30's FORMAT contains a carriage control 1X, but line 20's FORMAT omits the carriage control. Put a carriage control in formats that PRINT, but not in formats that READ.

On PDP computers, say ACCEPT instead of READ.

If you input a blank space, the computer treats it as a zero. For example, suppose you input:

```
-3 28219
```

Because of the I4 format, the computer will read just the first 4 characters, which are -3 2; the blank space between the 3 and the 2 is treated as a zero, so N will be -302.

Suppose you input:

```
57
```

Because of the I4 format, the computer will read the 5, the 7, and two blanks. Since the blanks are treated as zeros, N will be 5700.

Suppose you input:

```
■■■9527
```

Because of the I4 format, the computer will read the three beginning blanks and the 9. Since the blanks are treated as zeros, N will be 0009, which is 9. Line 30 makes the computer print:

```
   9
```

Suppose you write a program containing these lines:

```
      READ 20, L,M,N
20    FORMAT (I3,I4,2X,I2)
```

When you run that program, suppose you input:

```
581941389728 24
```

The I3 format makes the first 3 characters (581) be L; the I4 format makes the next 4 characters (9413) be M; the 2X format makes the next 2 characters (89) be skipped over; the I2 format makes the next two characters (72) be N; the remaining characters (824) are ignored. So the line is split like this:

```
variables, in the READ statement→→ | L  | M  |  | N |
needs, in the FORMAT statement←←   I3  I4   2X  I2
line you input→→→→→→→→→→→→→→→→→→→ |581|9413|89|72|824
```

Suppose you write a program containing these lines:

```
      READ 20, J,K,L,M,N
20    FORMAT (2I3)
```

The format says to read two 3-character integers on each line. Suppose you input:

```
78345692
85431684
46185327
```

J will be 783, and K will be 456. L will be 854, and M will be 316. N will be 461. That's how to input integers. Here's how to input real numbers instead....

```
      PRINT 10
10    FORMAT (1X,'TYPE SOME DIGITS')
      READ 20, P
20    FORMAT (F6.2)
      PRINT 30, P
30    FORMAT (1X,G14.6)
      END
```

The F6.2 format means: read 6 characters; if they don't contain the decimal point, insert it before the last 2 digits. For example, suppose you input:

```
327514968
```

The computer reads the first 6 characters (327514). Since they don't contain the decimal point, the computer inserts it before the last 2 digits, so P is 3275.14. Line 30 prints:

```
 3275.14
```

For that program, suppose you input:

```
7.5423967
```

The computer reads the first 6 characters (7.5423). Since they already contain the decimal point, F is 7.5423. Line 30 prints:

```
 7.54230
```

Suppose you input:

```
497E3
```

The computer reads 6 characters (497E3, followed by a blank). Since blanks are treated as zeros, the computer gets 497E30. Since 497E30 doesn't contain the decimal point, the computer inserts it before the last 2 digits, so P is 4.97E30. Line 30 prints:

```
 0.497000E+31
```

OMITTING FORMATS

Most computers let you omit numeric formats. This program works on most modern computers (such as computers having FORTRAN 77, PDP-20 computers, PDP-10 computer using FORTRAN 10, PDP-11 computers using FORTRAN IV-PLUS, CDC computers using FORTRAN IV-EXTENDED, and IBM computers using FORTRAN H-EXTENDED):

```
      PRINT 10                          On PDP computers, say TYPE instead of PRINT,
10    FORMAT (1X,'TYPE TWO INTEGERS')   and say ACCEPT instead of READ.
      READ *, M,N
      ISUM=M+N
      PRINT *, ISUM
      END
```

The word READ is followed by an asterisk, instead of a FORMAT number. The last PRINT is followed by an asterisk also. The asterisk makes the computer invent its own FORMAT. To make the program add 241 and 82976, input the numbers, separated by a comma:

```
241,82976
```

The computer will notice the comma's location and automatically use an I3 format for 241, a 1X format to skip over the comma, and an I5 format for 82976. To print ISUM, the computer will use a safe format, such as I14 or I15.

By omitting formats, you gain two advantages:

√ You can write FORTRAN programs more quickly.

√ The person who inputs doesn't have to worry about whether his spacing matches the format. The computer invents a format that matches his input.

On CDC computers using TS FORTRAN, and on Honeywell computers, omit the asterisk after READ and PRINT:

```
      PRINT 10
10    FORMAT (1X,'TYPE THE NUMBERS')
      READ, M,N
      ISUM=M+N
      PRINT, ISUM
      END
```

On PDP-10 computers using F40 FORTRAN, and on PDP-11 computers using regular FORTRAN IV, you need FORMATs, but omit the number in the I format:

```
      TYPE 10
10    FORMAT (1X,'TYPE THE NUMBERS')
      ACCEPT 20, M,N
20    FORMAT (1X,2I)
      ISUM=M+N
      TYPE 30, ISUM
30    FORMAT (1X,I)
      END
```

You can use similar short cuts for real numbers.

PRETTIER PROGRAMS

COMMENTS

If you type C instead of a line number, the computer will ignore the line.

```
      N=50+13
C  I HATE COMPUTERS!
      PRINT 10, N
10    FORMAT (1X,I14)
      END
```

The computer will ignore the Comment. The computer will print 63.

The C in FORTRAN is like the REMARK in BASIC: use it to document your program.

CONTINUATION

Some computers look at only the first 72 characters of each line of your program: if your line contains more than 72 characters, it won't work.

If you want to type a long statement, type just the first 72 characters. On the line below, type the remaining characters, beginning in the 7th print position; and in the 6th print position type a 6:

```
      PRINT 10
10    FORMAT (1X,'I LIKE ROSES IN MY TEA.'/1X,'THEY MAKE IT GLOW RED, LI
     6KE HOT BLOOD.')
      END
```

The computer will print:

```
I LIKE ROSES IN MY TEA.
THEY MAKE IT GLOW RED, LIKE HOT BLOOD.
```

A line that has a 6 in column 6 is called a *continuation line*, because it's a continuation of the line above it.

(I suggest you put a 6 in column 6. But some programmers put a 1 in column 6, or a 9 in column 6, or a + in column 6, or a * in column 6, or a $ in column 6. In fact, you can put *any character* in column 6, except a zero. Choose your favorite character; the computer doesn't care.)

On PDP computers, put the 6 immediately after a controlled I, instead of in column 6:

```
10    FORMAT (1X,'I LIKE ROSES IN MY TEA.'/1X,'THEY MAKE IT GLOW RED, LI
      6KE HOT BLOOD.')
```

(I suggest you put a 6 after the controlled I. But you can put *any non-zero digit*, instead of a 6.)

On CDC computers using TS FORTRAN, type a + instead of a 6, and put it immediately after the edit number, with no intervening spaces:

```
00120 10 FORMAT (1X,'I LIKE ROSES IN MY TEA.'/1X,'THEY MAKE IT GLOW RED, LI
00130+KE HOT BLOOD.')
```

FORTRAN allows the word IF:

Statement	Meaning
`IF (I .LT. 5) J=3`	If I is Less Than 5, let J=3.
`IF (I .LT. 5) GO TO 80`	If I is Less Than 5, go to line 80.
`IF (I .LT. 5) STOP`	If I is Less Than 5, stop.
`IF (I .LT. 5) PRINT 10, J`	If I is Less Than 5, print J by using line 10's FORMAT.

The condition must be enclosed in parentheses. LT must be enclosed in periods.

FORTRAN uses these clauses:

Clause	Meaning
`IF (I .LT. 5)`	if I is Less Than 5
`IF (I .GT. 5)`	if I is Greater Than 5
`IF (I .LE. 5)`	if I is Less than or Equal to 5
`IF (I .GE. 5)`	if I is Greater than or Equal to 5
`IF (I .EQ. 5)`	if I is EQual to 5
`IF (I .NE. 5)`	if I is Not Equal to 5

You can use the words AND and OR:

Idea	How to say it in FORTRAN
if I is 2 or 9 or 13	`IF (I .EQ. 2 .OR. I .EQ. 9 .OR. I .EQ. 13)`
if I is an integer from 1 to 100	`IF (I .GE. 1 .AND. I .LE. 100)`
if X<Y<Z	`IF (X .LT. Y .AND. Y .LT. Z)`
if A is less than both B and C	`IF (A .LT. B .AND. A .LT. C)`
if X is negative or between 5 and 9	`IF (X .LT. 0.0 .OR. X .GE. 5.0 .AND. X .LE. 9.0)`

Take this test: cover the column that says "How to say it in FORTRAN". Try to translate each "Idea" into FORTRAN, then check your answers. If one of your answers is shorter than the correct answer, take the test again! For example, the following answer to the first idea is *wrong*:

```
IF (I .EQ. 2 .OR. 9 .OR. 13)
```

FORTRAN 77 lets you say "END IF". For example, here's how FORTRAN 77 lets you say, "If I is greater than 5, let J be 80 and let K be 90":

```
IF (I .GT. 5) THEN
   J=80
   K=90
END IF
```

Here's how FORTRAN 77 lets you say, "If I is greater than 5, let J be 80 and let K be 90; but if I is *not* greater than 5, let J be 30 and let K be 50":

```
IF (I .GT. 5) THEN
   J=80
   K=90
ELSE
   J=30
   K=50
END IF
```

Here's how FORTRAN 77 lets you say, "If I is greater than 5, let J be 80 and let K be 90; if I is *not* greater than 5, but I is greater than 2, let J be 81 and let K be 92; if I is not greater than 2, let J be 30 and let K be 50":

```
IF (I .GT. 5) THEN
   J=80
   K=90
ELSE IF (I .GT. 2) THEN
   J=81
   K=92
ELSE
   J=30
   K=50
END IF
```

Warning: to say "END IF", you must get FORTRAN 77. If you use FORTRAN IV instead, "END IF" doesn't work. I recommend that you get FORTRAN 77.

Here's a different kind of IF statement:

```
IF (X) 20,50,90
```

It means:

if X is a negative number, go to line 20;
if X is zero, go to line 50;
if X is a positive number, go to line 90.

That kind of IF statement is called a *three-way IF*, or an *arithmetic IF*. (To pronounce "arithmetic", put the accent on met.) The other kind of IF is called a *logical IF*.

COMPUTED GO TO

In your program, you can say:

```
GO TO (80,100,20,350), I
```

That means: go to either 80, 100, 20, or 350, depending on what I is. More specifically, it means:

go to line 80, if I is 1;
go to line 100, if I is 2;
go to line 20, if I is 3;
go to line 350, if I is 4;
proceed to the line underneath, if I is a different integer.

That FORTRAN statement is called a *computed GO TO*. It resembles this BASIC statement:

```
30 ON I GO TO 80,100,20,350
```

This program prints the square of every number from 80 to 100, and then prints GET LOST:

BASIC

```
10 FOR I = 80 TO 100
20     PRINT I↑2
30 NEXT I
40 PRINT "GET LOST"
```

FORTRAN

```
      DO 30 I=80,100
         J=I**2
         PRINT 20, J
20       FORMAT (1X,I14)
30    CONTINUE
      PRINT 40
40    FORMAT (1X,'GET LOST')
      END
```

If you compare the BASIC with the FORTRAN, you'll notice FORTRAN uses the word DO instead of FOR, uses a comma instead of TO, and uses CONTINUE instead of NEXT. The statement *DO 30 I=5,9* means: DO every line up through line 30, repeatedly, as I goes from 5 to 9. In BASIC, programmers indent every line between FOR and NEXT; in FORTRAN, programmers indent every line between DO and CONTINUE. In BASIC, the indented lines are called a *FOR...NEXT loop*; in FORTRAN, they're called a *DO loop*.

If you want the computer to print the square of every fifth number from 80 to 100, change the top line of the program:

BASIC

```
10 FOR I = 80 TO 100 STEP 5
```

FORTRAN

```
DO 30 I=80,100,5
```

In a DO statement, some computers allow only positive integer variables and constants

Not allowed	Why	What to say instead
`DO 10 X=1.0,5.0`	reals are not allowed	`DO 10 I=1,5`
`DO 10 X=17.3,98.5`	reals are not allowed	`DO 10 I=173,985` `X=I/10.0`
`DO 10 I=0,5`	0 is not positive	`DO 10 J=1,6` `I=J-1`
`DO 10 I=-3,5`	-3 is not positive	`DO 10 J=1,9` `I=J-4`
`DO 10 I=100,7,-1`	-1 is not positive	`DO 10 J=7,100` `I=107-J`
`DO 10 I=5,J+K`	+ is not allowed	`L=J+K` `DO 10 I=5,L`

Other computers are more generous and allow anything. Find out about yours.

If you say—

```
DO 10 I=1,N
```

the computer will do up through line 10, N times. For example, if N is 73, the computer will do up through line 10, 73 times. If N is 2, the computer will do up through line 10, twice. If N is 1, the computer will do up through line 10, once.

What happens in N is less than 1? The answer depends on which version of FORTRAN you're using. If you're using FORTRAN 77, the computer will skip the loop, and proceed to the line below line 10. But if you're using FORTRAN IV, the computer will do the loop once, as if N were 1. FORTRAN 77 makes more sense, but at the moment most computers are still using FORTRAN IV.

In the middle of a DO loop, don't change the value of the index. For example, if your DO loop begins with—

```
DO 10 I=1,100
```

don't insert this line in the middle of your loop:

```
I=14
```

It will confuse the computer.

LISTS

SUBSCRIPTS

Like BASIC, FORTRAN permits subscripts:

```
      DIMENSION X(4)
      X(1)=.21
      X(2)=.3
      X(3)=1.08
      X(4)=5.0
      SUM=X(1)+X(2)+X(3)+X(4)
      PRINT 10, X,SUM
10    FORMAT (1X,G14.6)
      END
```

X will be a list of 4 numbers, called X(1), X(2), X(3), and X(4). Since X doesn't begin with I, J, K, L, M, or N, the 4 numbers will be real.

The computer will print the list and the SUM, like this:

```
 0.210000
 0.300000
  1.08000
  5.00000
  6.59000
```

If you change the format to (1X,5G14.6), the computer will print all 5 numbers on the same line:

```
 0.210000      0.300000      1.08000       5.00000       6.59000
```

You must say DIMENSION if your program uses subscripts, even if the subscripts are small. Say DIMENSION at the very top of the program. Make sure you say DIMENSION, not DIM or DIMENSIONS. The computer assumes all subscripts will be positive, so don't say X(0). If a subscript is zero or negative or larger than the DIMENSION statement says, the computer might not notice your error, and will print wrong answers without warning you.

If your program begins like this—

```
      DIMENSION A(6)
      READ 10, A
```

the computer will begin by reading 6 real numbers, which will become A(1), A(2), A(3), A(4), A(5), and A(6).

If you want T to be a table of numbers, and you want T to have 4 rows and 2 columns, begin your program by saying:

```
      DIMENSION T(4,2)
```

To print T, say:

```
      PRINT 10, T(1,1), T(1,2)
      PRINT 10, T(2,1), T(2,2)
      PRINT 10, T(3,1), T(3,2)
      PRINT 10, T(4,1), T(4,2)
10    FORMAT (1X,2G14.6)
```

If you say—

```
      PRINT 10, T
```

the computer will print the entire table T, but in an undesirable order: it will print T(1,1), T(2,1), T(3,1), and T(4,1), then T(1,2), T(2,2), T(3,2), and T(4,2). Similarly, "READ 5, T" makes the computer read T in an undesirable order.

This statement—

```
PRINT 10, X(3),X(4),X(5),X(6),X(7)
```

can be written more briefly, like this:

```
PRINT 10, (X(I), I=3,7)
```

It means: using line 10's format, print the value of X(I), for I = 3 to 7. The construction (X(I), I=3,7) is called an *implied DO loop*. Notice the parenthesis at the beginning, the parenthesis at the end, and the commas. Here are other examples of implied DO loops:

Implied DO loop	Meaning
(X(I), I=100,120,5)	X(100),X(105),X(110),X(115),X(120)
(X(I),Y(I), I=3,7)	X(3),Y(3), X(4),Y(4), X(5),Y(5), X(6),Y(6), X(7),Y(7)

Here's a calendar:

```
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

This program prints it:

```
      PRINT 10, (I, I=1,31)
10    FORMAT (1X,I2,1X,I2,1X,I2,1X,I2,1X,I2,1X,I2,1X,I2)
      END
```

The program's top statement says to print every value of I, for I = 1 to 31. The FORMAT says to print 7 integers on each line, and separate the integers by spaces. Since 1X followed by I2 is about the same as I3, you can write the FORMAT more briefly:

```
10    FORMAT (I3,I3,I3,I3,I3,I3,I3)
```

Even more briefly:

```
10    FORMAT (7I3)
```

If T is a table having 3 rows and 5 columns, these lines will print it in the correct order:

```
      PRINT 20, T(1,1), T(1,2), T(1,3), T(1,4), T(1,5)
      PRINT 20, T(2,1), T(2,2), T(2,3), T(2,4), T(2,5)
      PRINT 20, T(3,1), T(3,2), T(3,3), T(3,4), T(3,5)
20    FORMAT (1X,5G14.6)
```

Tbis short cut does the same thing:

```
      PRINT 20, (T(1,J), J=1,5)
      PRINT 20, (T(2,J), J=1,5)
      PRINT 20, (T(3,J), J=1,5)
20    FORMAT (1X,5G14.6)
```

Here's a shorter cut:

```
      PRINT 20, ((T(I,J), J=1,5), I=1,3)
20    FORMAT (1X,5G14.6)
```

To read the table, say:

```
      READ 10, ((T(I,J), J=1,5), I=1,3)
```

DATA

This example shows FORTRAN's DATA statement, which differs from BASIC's:

```
      DATA X/8.7/, Y/1.4/, Z/9.0/     X is 8.7, Y is 1.4, and Z is 9.0.
      X=100.6                         X changes to 100.6.
      PRINT 10, X,Y,Z                 The computer will print:
10    FORMAT (1X,3G14.6)                 100.600       1.40000       9.00000
      END
```

In that DATA statement, you must write 9.0, not 9: the number must be real, since Z is real.

The DATA statement resembles these three statements—

```
X=8.7
Y=1.4
Z=9.0
```

but is faster.

Here's another way to type the DATA statement:

```
DATA X,Y,Z/8.7,1.4,9.0/
```

It says X, Y, and Z are 8.7, 1.4, and 9.0 respectively.

Like the DIMENSION statement, the DATA statement belongs at the very top of the program. If you want both a DIMENSION statement and a DATA statement, put the DIMENSION statement first.

This DATA statement says A, B, C, D, and E are all 1.7, and X, Y, and Z are all 9.6:

```
DATA A,B,C,D,E,X,Y,Z/1.7,1.7,1.7,1.7,1.7,9.6,9.6,9.6/
```

Since the first 5 numbers are 1.7, and the next 3 numbers are 9.6, you can write more briefly:

```
DATA A,B,C,D,E,X,Y,Z/5*1.7,3*9.6/
```

To make A be this list—

```
81.7
92.6
25.3
49.8
72.1
68.8
```

begin your program with these lines:

```
DIMENSION A(6)
DATA A/81.7,92.6,25.3,49.8,72.1,68.8/
```

To make T be this table—

```
8.4  9.7
5.1  6.8
2.5  7.2
6.3  9.8
```

begin your program with these lines:

```
DIMENSION T(4,2)
DATA T/8.4,5.1,2.5,6.3,9.7,6.8,7.2,9.8/
```

Notice you must list the entire first column, then the second.

This program works on most computers:

```
      N='UP'
      PRINT 10, N
10    FORMAT (1X,A2)
      END
```

The top line says N is the string 'UP'. The next line says to print N, using the format in line 10. The A2 format means a 2-character string. (The A is derived from the word Alphabet.) The computer will print:

```
UP
```

Some computers allow apostrophes only in FORMAT, DATA, and CALL statements. (You'll learn about CALL statements later.) On such computers, the statement N='UP' is illegal. Instead say:

```
      DATA N/'UP'/
```

A string statement should look like an integer: it should begin with the letter I, J, K, L, M, or N. You can say N='UP' but shouldn't say X='UP'.

The string a variable stands for must be short. You can say N='UP' but not N='SUPERCALIFRAGILISTICEXPIALIDOCIOUS'. The longest permissible string depends on your computer:

Computer	Longest permissible string
PDP-11 (using FORTRAN IV)	2 characters
PDP-11 (using FORTRAN IV-PLUS), IBM	4 characters
PDP-10, PDP-20, Honeywell	5 characters
CDC	10 characters

Example:

```
      PRINT 10
10    FORMAT (1X,'DO YOU LIKE ME?')
      READ 20, IREPLY
20    FORMAT (A1)
      IF (IREPLY .EQ. 'Y') PRINT 30
30    FORMAT (1X,'I LIKE YOU TOO!')
      PRINT 40
40    FORMAT (1X,'SO LONG, BUSTER.')
      END
```

The first pair of lines make the computer print DO YOU LIKE ME? The next pair set IREPLY equal to the first letter the human types. If the human types YESIREE, the computer will set IREPLY equal to 'Y' and will therefore print:

```
I LIKE YOU TOO!
SO LONG, BUSTER.
```

But if the human types NOT AT ALL, the computer will set IREPLY equal to 'N' and will therefore print just:

```
SO LONG, BUSTER.
```

In that program, the string is called IREPLY instead of REPLY, to make it an integer.

Advanced example:

```
      DIMENSION NAME(25)
      PRINT 10
10    FORMAT (1X,'WHAT IS YOUR NAME?')
      READ 20, NAME
20    FORMAT (25A2)
      PRINT 30, NAME
30    FORMAT (1X,'I HATE ANYONE NAMED ',25A2)
      END
```

The top line says NAME will be a list of 25 elements. The next pair of lines print WHAT IS YOUR NAME? If the human answers—

```
BARTHOLOMEW HIERONYMOUS MCGILLICUDDY, M.D.
```

the format in line 20 sets NAME equal to 25 two-character strings:

```
NAME(1) is 'BA'          NAME(10) is 'YM'          NAME(19) is ', '
NAME(2) is 'RT'          NAME(11) is 'OU'          NAME(20) is 'M.'
NAME(3) is 'HO'          NAME(12) is 'S '          NAME(21) is 'D.'
NAME(4) is 'LO'          NAME(13) is 'MC'          NAME(22) is '  '
NAME(5) is 'ME'          NAME(14) is 'GI'          NAME(23) is '  '
NAME(6) is 'W '          NAME(15) is 'LL'          NAME(24) is '  '
NAME(7) is 'HI'          NAME(16) is 'IC'          NAME(25) is '  '
NAME(8) is 'ER'          NAME(17) is 'UD'
NAME(9) is 'ON'          NAME(18) is 'DY'
```

Format 30 prints:

```
I HATE ANYONE NAMED BARTHOLOMEW HIERONYMOUS MCGILLICUDDY, M.D.
```

To make that program run faster, use fewer strings. For example, if you have a CDC computer, each string can be as long as 10 characters, so you need only 5 strings to make 50 characters:

```
→→      DIMENSION NAME(5)
        PRINT 10
  10    FORMAT (1X,'WHAT IS YOUR NAME?')
        READ 20, NAME
→→20    FORMAT (5A10)
        PRINT 30, NAME
→→30    FORMAT (1X,'I HATE ANYONE NAMED ',5A10)
        END
```

On computers having FORTRAN 77, and on IBM computers using WATFIV, you can request super-long strings:

```
→→      CHARACTER*50 NAME
        PRINT 10
  10    FORMAT (1X,'WHAT IS YOUR NAME?')
        READ 20, NAME
→→20    FORMAT (A50)
        PRINT 30, NAME
→→30    FORMAT (1X,'I HATE ANYONE NAMED ',A50)
        END
```

The top line requests that NAME be a 50-character string. Like the DIMENSION statement, the CHARACTER statement must be put at the very top of the program, above even the DATA statements.

FANCY FORMATS

DOUBLE APOSTROPHE

To make the computer print an apostrophe, use two apostrophes next to each other.

```
      PRINT 10
10    FORMAT (1X,'MOMMY ISN''T HERE')
      END
```

The computer will print:

```
MOMMY ISN'T HERE
```

T FORMAT

You saw this program for meatball lovers:

```
      PRINT 10
10    FORMAT (1X,'EAT A',8X,'MEATBALL')
      END
```

It made the computer print EAT A, then 8 blank spaces, then MEATBALL:

```
EAT A        MEATBALL
```

This program does the same thing:

```
      PRINT 10
10    FORMAT (1X,'EAT A',T15,'MEATBALL')
      END
```

It makes the computer print EAT A, then Tab over to the 15th position on the line, and print MEATBALL. When the computer tabs to the 15th position, it considers the carriage control to be the first "position"; the E in EAT is the first character the computer will print, and the computer considers it to be the second "position"; the A in EAT is the second character the computer will print, and the computer considers it to be the third "position"; the M in MEATBALL is the 14th character the computer will print, since the T15 says it is the 15th "position".

BLANK RECORDS

Example:

```
      PRINT 10
10    FORMAT (1X,'PLEASE'/1X,'NIBBLE'//1X,'MY'/////1X,'CHEESE')
      END
```

Line 10 makes the computer print several lines. The first line will say PLEASE. The next line will say NIBBLE. The next line will be blank. The next line will say MY. The next 4 lines will be blank. The last line will say CHEESE. So altogether, the computer will print:

```
PLEASE
NIBBLE

MY




CHEESE
```

REPEATED FORMATS

The computer can feel very depressed:

```
      PRINT 10
10    FORMAT (1X,'I FEEL ',3('DOWN'))
      END
```

Line 10 is an abbreviation for this format:

```
1X,'I FEEL ','DOWN','DOWN','DOWN'
```

The computer will print:

```
I FEEL DOWNDOWNDOWN
```

Let's burp and pray:

```
      PRINT 10
10    FORMAT (1X,'JACK ',2('BURPS'/1X,'MARY '),'ALSO PRAYS')
      END
```

Line 10 is an abbreviation for this format:

```
1X,'JACK ','BURPS'/1X,'MARY ','BURPS'/1X,'MARY ','ALSO PRAYS'
```

The computer will print:

```
JACK BURPS
MARY BURPS
MARY ALSO PRAYS
```

E FORMAT

For real numbers, the usual formats are F and G, but another option is E. If you say E14.6 instead of G14.6, the computer will print an E in the answer. Here are examples:

Using G14.6 format	Using E14.6 format
▮-0.283941E-29	▮-0.283941E-29
▮▮0.293027▮▮▮▮	▮▮0.293027E+00
▮▮▮5.34523▮▮▮▮	▮▮0.534523E+01
▮▮▮39.4539▮▮▮▮	▮▮0.394539E+02
▮▮▮47802.3▮▮▮▮	▮▮0.478023E+05
▮▮▮986327.▮▮▮▮	▮▮0.986327E+06
▮▮0.288341E+24	▮▮0.288341E+24

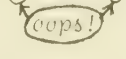
The G14.6 format is easier for a human to read than E14.6. But most programmers are stupid, don't know about G14.6, and use E14.6 instead.

P FORMAT

FORTRAN's notation differs from BASIC. If you ask the computer to print 288341000000000000000000.0 in FORTRAN, by using G14.6 (or E14.6), the computer will normally print a 0 before the decimal point, like this: 0.288341E+24. In BASIC, the computer will print a non-zero digit before the decimal point, like this: 2.88341E+23.

If you're writing a program in FORTRAN, but you prefer BASIC's notation, ask for 1PG14.6 (or 1PE14.6). The 1P makes the computer imitate BASIC. But if a FORMAT contains 1PG, it must not contain F afterwards; this will print a wrong answer:

```
FORMAT (1X,1PG14.6,F8.2)
```



FUNCTIONS

SQUARE ROOT

This program finds the square root of 9:

```
      A=SQRT(9.0)
      PRINT 10, A
10    FORMAT (1X,G14.6)
      END
```

The computer will print:

```
 3.00000
```

In that program, you must say SQRT(9.0), not SQRT(9). If you say SQRT(9), the computer will either gripe or print a wrong answer.

The number in parentheses must be real. That's why you can say SQRT(9.0) but not SQRT(9). You can say SQRT(8.0+1.0) but not SQRT(8+1). You can say SQRT(X) but not SQRT(J).

Be careful when you translate from BASIC to FORTRAN: in BASIC, you say SQR; in FORTRAN, you say SQRT instead.

This program prints a table, showing the square root of 2.0, the square root of 3.0, the square root of 4.0, the square root of 5.0, etc.:

```
      X=2.0
10    Y=SQRT(X)
      PRINT 20, X,Y
20    FORMAT (1X,2G14.6)
      X=X+1.0
      GO TO 10
      END
```

The computer will print:

```
 2.00000        1.41421
 3.00000        1.73205
 4.00000        2.00000
 5.00000        2.23607
 6.00000        2.44949
 7.00000        2.64575
 8.00000        2.82843
 9.00000        3.00000
 10.0000        3.16228
 etc.
```

FLOAT

If you say FLOAT, the computer will create a FLOATing-point number (in other words, a real number), by using an integer. For example, FLOAT(3) is 3.0. If J is 7, then FLOAT(J) is 7.0.

The word FLOAT can help you reach the following goals....

GOAL: find the square root of J.
DIFFICULTY: you aren't allowed to say SQRT(J), because what you take the square root of must be a real number; you can say SQRT(X) but not SQRT(J).
SOLUTION: say X=J, and then say SQRT(X).
SHORTER SOLUTION: say SQRT(FLOAT(J)).

GOAL: divide I by J accurately.
DIFFICULTY: saying I/J doesn't solve the problem, because when the computer divides integers it gives an integer answer, instead of an accurate real answer.
SOLUTION: say X=I, then Y=J, then X/Y.
SHORTER SOLUTION: say FLOAT(I)/FLOAT(J).

RANDOM NUMBERS

Here's how to set R equal to a random decimal between 0 and 1:

Computer	What to say
CDC	R=RANF(0)
PDP-10, PDP-20	R=RAN(0)
PDP-11	R=RAN(ISEED,ISEED2)

To randomize the random numbers on PDP-10 and PDP-20 computers, put these lines near the top of your program:

```
CALL TIME(ISEED,ISEED2)
CALL SETRAN(MOD(ISEED/2+ISEED2/2, 2147483648))
```

On other computers, randomizing is even more complicated; ask the people who run your computer center.

MAXIMA AND MINIMA

To find the maximum real number in a list, ask for AMAX1. For example, AMAX1(4.7, 2.8, 41.6, 9.2, 82.3, 9.7) is 82.3. And AMIN1(4.7, 2.8, 41.6, 9.2, 82.3, 9.7) is the minimum, which is 2.8. If the numbers in the list are integers, say MAX0 instead of AMAX1, and say MIN0 instead of AMIN1.

zero, not oh

ABSOLUTE VALUE

ABS(X) means the ABSolute value of X; in other words, X without its minus sign. For example:

ABS(-5.2) is 5.2
ABS(-7.0) is 7.0
ABS(9.3) is 9.3

For integers, say IABS instead of ABS. For example, IABS(-7) is -7.

REMAINDER

When you divide 11 by 4, the remainder is 3:

```
  2
4)11
 -8
  3 is the remainder
```

If you ask for MOD(11,4) the computer will divide 11 by 4 and get the remainder, which is 3; so MOD(11,4) is 3.

Use MOD for integers; use AMOD for reals. For example, AMOD(11.0, 4.0) is 3.0.

If your computer has FORTRAN 77, or your computer is an IBM having FORTRAN H-EXTENDED, or your computer is a PDP-11 having FORTRAN IV-PLUS, you can use these trigonometric functions:

Symbol	Meaning
SIN(X)	the SINe of X radians
COS(X)	the COSine of X radians
TAN(X)	the TANgent of X radians
ASIN(X)	the ArcSINe of X in radians; the number whose sine is X
ACOS(X)	the ArcCOSine of X in radians; the number whose cosine is X
ATAN(X)	the ArcTANgent of X in radians; the number whose tangent is X
SINH(X)	the SINe Hyperbolic of X
COSH(X)	the COSine Hyperbolic of X
TANH(X)	the TANgent Hyperbolic of X

If your computer is old-fashioned, it restricts you:

Old-fashioned computer	Restriction
PDP-10, PDP-20	you can't say TAN(X)
CDC	you can't say SINH(X) or COSH(X)
other IBM computers	say ARSIN(X), not ASIN(X); say ARCOS(X), not ACOS(X)
FORTRAN IV computers	you can't say TAN(X), ASIN(X), ACOS(X),SINH(X), or COSH(X)

You can replace X by any *real* number. For example, you can say SIN(4.0) but not SIN(4); you can say SIN(Y) but not SIN(J).

Be careful when you translate from BASIC to FORTRAN: in BASIC, you say ATN; in FORTRAN, you say ATAN instead.

You've seen that SIN(X) is the sine of X *radians*. But what's the sine of X *degrees*? On PDP-10 and PDP-20 computers, you can find the sine of X degrees by asking for SIND(X); and you can find the cosine of X degrees by asking for COSD(X).

ATAN2(Y,X) is about the same as ATAN(Y/X), but is faster, more accurate, and gives a useful answer even when X is zero or negative. ATAN2(Y,X) is the angle (in radians) of the line that goes through the origin and the point (X,Y).

You can use these functions:

Function	Meaning
EXP(X)	e^X
ALOG(X)	$\log_e X$
ALOG10(X)	$\log_{10} X$

You can replace X by any real number, but not by an integer. Each of those functions produces a *real* answer, since none of them begins with I, J, K, L, M, or N. The logarithm function is called ALOG instead of LOG, to avoid beginning with L. Be careful when you translate from BASIC to FORTRAN: in BASIC, you say LOG; in FORTRAN, you say ALOG instead.

EXOTIC FEATURES

DOUBLE PRECISION

Some computers are more accurate than others:

Computer	Accuracy for real numbers
PDP-11, IBM	7 digits
PDP-10, PDP-20, Honeywell	8 digits
CDC	14 digits

For example, suppose you feed this program to a PDP-11 or IBM:

```
      A=5398.1642376236
      PRINT 10, A
10    FORMAT (1X,F15.10)
      END
```

Expect the first 7 digits the computer prints to be correct (5398.164), but the remaining digits it prints to be wrong: they arise from round-off error inside the computer.

A PDP-11 or IBM prints *some* numbers to an accuracy of 8 digits and others to an accuracy of only 6 digits, but 7 digits is typical.

For real numbers, I recommend you use a G14.6 format, because it's safe: it prints just the first 6 digits. If you want to see further digits that are probably correct, use these formats instead:

Computer	Format	What the format does
PDP-11, IBM	G15.7	prints the first 7 digits
PDP-10, PDP-20, Honeywell	G16.8	prints the first 8 digits
CDC	G22.14	prints the first 14 digits

You can obtain extra accuracy, by requesting *double precision*:

Computer	Accuracy for double precision numbers	Format
PDP-11	14 digits	D22.14
PDP-10 (using KA), Honeywell	16 digits	D24.16
IBM	17 digits	D25.17
PDP-10 (using KI or KL), PDP-20	18 digits	D26.18
CDC	29 digits	D37.29

Each of these programs computes the square root of $6.3 \cdot 10^8$ on a PDP-11 computer:

Using reals

```
      A=SQRT(6.3E8)
      PRINT 10, A
10    FORMAT (1X,G15.7)
      END
```

Using double precision

```
      DOUBLE PRECISION A
      A=DSQRT(6.3D8)
      PRINT 10, A
10    FORMAT (1X,D22.14)
      END
```

The program on the left computes 7 digits. The program on the right computes 14. Comparing the programs, you'll notice four differences:

1. For double precision, you must use double precision numbers. Instead of saying 6.3E8, say 6.3D8. The D means Double precision.

Real number	Double precision number
6.3E8	6.3D8
29.6	29.6D0

↖ *this is a zero*

2. For double precision, you must use double precision functions. Instead of saying SQRT, say DSQRT.

Real function	Double precision func.
SQRT	DSQRT
AMAX1	DMAX1
ABS	DABS
AMOD	DMOD
EXP	DEXP
ALOG	DLOG
ALOG10	DLOG10

Real function	Double precision func.
SIN	DSIN
COS	DCOS
TAN	DTAN
ASIN	DASIN
ACOS	DACOS
ATAN	DATAN
ATAN2	DATAN2
SINH	DSINH
COSH	DCOSH
TANH	DTANH

3. For double precision, you must use double precision variables. Normally, the variable A would be real; to make it double precision instead, say:

```
DOUBLE PRECISION A
```

Normally, the variables LENGTH and MASS would be integers, and VOLUME would be real; to make them all double precision, say:

```
DOUBLE PRECISION LENGTH,MASS,VOLUME
```

Like the DIMENSION and CHARACTER statements, the DOUBLE PRECISION statement must be put at the very top of the program, above even the DATA statements. If you say—

```
IMPLICIT DOUBLE PRECISION(D)
```

every variable whose first letter is D will automatically be double precision: for example, DISTAN and DIAMET and DSIZE will automatically be double precision.

4. For double precision you must use double precision formats. Instead of a G format, use D22.14, or whichever D format is appropriate for your computer.

Although double precision arithmetic is more precise than real arithmetic, it's also more expensive: it consumes more of the computer's time, and the numbers consume more of the computer's memory.

If you combine an integer or a real number with a double precision number, the answer will be double precision.

In mathematics, the square root of -1 is called i. So i^2 is -1. The number i obeys most of the rules of algebra:

i+i is 2i
2i+3i is 5i
(8+2i) + (7+3i) is 15+5i
(8+2i) * (7+3i) is 8*7 + 8*3i + 2i*7 + 2i*3i, which is 56 + 24i + 14i + $6i^2$, which is 56 + 24i + 14i + -6, which is 50+38i.

The number i is neither positive nor negative nor zero; it is pictured instead as being above the real number line:

```
                i

-5 -4 -3 -2 -1  0  1  2  3  4  5
```

2i is further above the real number line: it is 2 units above 0. Another example: 5+2i is 2 units above 5.

A number that involves i is called *complex*. So 5+2i is complex. Its *real part* is 5, its *imaginary part* is 2, and its *conjugate* is 5-2i.

This program multiplies 8+2i by 7+3i and prints the correct answer, 50+38i:

```
      COMPLEX B
      B=(8.0, 2.0) * (7.0, 3.0)
      PRINT 10, B
10    FORMAT (1X,2G14.6)
      END
```

FORTRAN says (8.0, 2.0) instead of 8+2i; the decimal points and parentheses are required. To make B complex instead of real, say COMPLEX B; like the DIMENSION and CHARACTER and DOUBLE PRECISION statements, the COMPLEX statement must be put at the very top of the program, above even the DATA statements. To print B, use the G14.6 format twice (once for the real part, and once for the imaginary part). The computer will print:

```
 50.0000        38.0000
```

If you say IMPLICIT COMPLEX(C), every variable whose first letter is C will automatically be complex.

Although you can write 8+2i as (8.0, 2.0), you cannot write X+Yi as (X, Y); instead write CMPLX(X, Y). The rule is: to build a complex number from variables instead of from constants, say CMPLX. The variables you use to build it must be real.

If B is complex, you can use these functions:

Function	Meaning
CSQRT(B)	the complex number that's the square root of B
CSIN(B)	the complex number that's the sine of B
CCOS(B)	the complex number that's the cosine of B
CEXP(B)	the complex number that's e^B
CLOG(B)	the complex number that's log B
CABS(B)	the *real* number that's the absolute value of B
REAL(B)	the *real* number that's the real part of B
AIMAG(B)	the *real* number that's the imaginary part of B
CONJG(B)	the complex number that's the conjugate of B

This program finds the square root of -9 and prints the correct answer, 3i:

```
      COMPLEX B,Z
      B=-9
      Z=CSQRT(B)
      PRINT 10, Z
10    FORMAT (1X,2G14.6)
      END
```

Since the top line says B is complex, the second line makes B the complex number -9+0i. The next line makes Z the square root of -9+0i, which is 0+3i. The computer will print the 0 and the 3:

```
 0.000000        3.00000
```

Since that program sets B equal to -9+0i, which FORTRAN writes as (-9.0, 0.0), you can shorten the program to this:

```
      COMPLEX Z
      Z=CSQRT( (-9.0, 0.0) )
      PRINT 10, Z
10    FORMAT (1X,2G14.6)
      END
```

Make sure the number inside CSQRT's parentheses is complex: you need the decimal points, comma, and parentheses.

When the computer combines a complex number with an integer, a real, or a double precision number, the result is complex.

Some computers limit your use of complex numbers, by restrictions such as:

"Don't say a complex number equals a double precision number",
"Don't combine a complex number with a double precision number",
"Don't raise a number to a complex power",
"Don't raise a complex number to a power, unless the power is an integer".

Find out whether your computer has those restrictions.

This program is a combination of two *routines*:

```
      PRINT 10
10    FORMAT (1X,'KIDS')
      CALL YUMMY
      PRINT 20
20    FORMAT (1X,'LOLLIPOPS')
      END

      SUBROUTINE YUMMY
      PRINT 10
10    FORMAT (1X,'SUCK')
      RETURN
      END
```

Each routine ends with the word END. The first routine is called the *main routine*; the second routine is called the *subroutine*, and begins with the word SUBROUTINE; I decided to name it YUMMY, so its top line says SUBROUTINE YUMMY. Above the word SUBROUTINE, I put a blank line. The blank line is optional; it helps humans find where the subroutine begins.

In the main routine, the top pair of lines makes the computer print:

```
KIDS
```

The next line says CALL YUMMY, which makes the computer skip down to subroutine YUMMY. Subroutine YUMMY makes the computer print—

```
SUCK
```

and then RETURN to the main routine, which finishes by printing:

```
LOLLIPOPS
```

Altogether, the program prints:

```
KIDS
SUCK
LOLLIPOPS
```

Notice that line 10 in the main routine is different from line 10 in the subroutine. Similarly, an X in the main routine is different from one in the subroutine:

```
      X=3.4
      CALL FUNNY
      PRINT 10, X
10    FORMAT (1X,G14.6)
      END

      SUBROUTINE FUNNY
      X=925.1
      Y=X+1.0
      PRINT 10, Y
10    FORMAT (1X,G14.6)
      RETURN
      END
```

The computer will set the main routine's X equal to 3.4. Then it will call FUNNY. The X in FUNNY is 925.1, so Y is 926.1, and the computer will print:

```
 926.100
```

When the computer returns to the main routine, it will print the main routine's X, which is still:

```
 3.40000
```

To pass information from one routine to another, put the information in parentheses:

```
      A=5.2
      CALL LEMON(A)
      PRINT 10, A
10    FORMAT (1X,G14.6)
      END

      SUBROUTINE LEMON(X)
      PRINT 100, X
100   FORMAT (1X,G14.6)
      X=7.1
      RETURN
      END
```

The computer sets A equal to 5.2. Then it calls LEMON. The A and X in parentheses mean: the main routine's A is the subroutine's X, so the subroutine's X is 5.2. Line 100 prints:

```
5.20000
```

The next line changes X to 7.1. When the computer returns to the main routine, the main routine's A is the subroutine's X, so the main routine's A is 7.1. Line 10 prints:

```
7.10000
```

A harder example:

```
      P=2.1
      CALL JUNK(P)
      PRINT 10, P
10    FORMAT (1X,G14.6)
      END

      SUBROUTINE JUNK(P)
      P=3.0*P
      RETURN
      END
```

The computer sets P equal to 2.1 and then calls JUNK. The P in parentheses means: the main routine's P is the subroutine's P. The subroutine tripes P, so P becomes 6.3. When the computer returns to the main routine, line 10 prints:

```
6.30000
```

Another example:

```
      Q=1.4
      CALL FAT(5.0, Q+.3, R)
      PRINT 10, R
10    FORMAT (1X,G14.6)
      END

      SUBROUTINE FAT(X, Y, Z)
      Z=X+Y
      RETURN
      END
```

When the main routine calls FAT, the subroutine's X is 5.0; Y is Q+.3, which is 1.7; and Z is R, which is undefined. The subroutine sets Z equal to X+Y, which is 6.7. When the computer returns to the main routine, the main routine's R is the subroutine's Z, which is 6.7. Line 10 prints:

```
6.70000
```

In that CALL, you must say 5.0, not 5, since X must be real. Saying 5 will confuse the computer and make it print a wrong answer.

This subroutine finds the sum and average of three real numbers— X, Y, and Z:

```
      SUBROUTINE STAT(X, Y, Z, SUM, AVERAG)
      SUM=X+Y+Z
      AVERAG=SUM/3.0
      RETURN
      END
```

This main routine uses STAT to find the sum and average of 8.1, 2.6, and 9.3:

```
      CALL STAT(8.1, 2.6, 9.3, SUM, AVERAG)
      PRINT 10, SUM,AVERAG
10    FORMAT (1X,2G14.6)
      END
```

This subroutine finds the sum and average of three *double precision* numbers:

```
      SUBROUTINE DSTAT(X, Y, Z, SUM, AVERAG)
      DOUBLE PRECISION X, Y, Z, SUM, AVERAG
      SUM=X+Y+Z
      AVERAG=SUM/3D0
      RETURN
      END
```

This main routine uses DSTAT to find the sum and average of π, e, and my phone number (2668128):

```
      DOUBLE PRECISION SUM, AVERAG
      CALL DSTAT(3.1415926535898D0, 2.7182818284590D0, 2668128D0, SUM,
     6AVERAG)
      PRINT 10, SUM, AVERAG
10    FORMAT (1X,2D22.14)
      END
```

You must say DOUBLE PRECISION in both the main routine and the subroutine.

This subroutine finds the sum and average of X_1, X_2, X_3, ..., X_N:

```
      SUBROUTINE STAT2(X, N, SUM, AVERAG)
      DIMENSION X(N)
      SUM=0.0
      DO 10 I=1,N
         SUM=SUM+X(I)
10    CONTINUE
      AVERAG=SUM/N
      RETURN
      END
```

This main routine uses STAT2 to find the sum and average of 8.4, 9.6, 20.1, 7.2, 91.5, and 3.6:

```
      DIMENSION X(6)
      DATA X/8.4,9.6,20.1,7.2,91.5,3.6/
      CALL STAT2(X, 6, SUM, AVERAG)
      PRINT 10, SUM,AVERAG
10    FORMAT (1X,2G14.6)
      END
```

You must put the DIMENSION statement in both the main routine and the subroutine. In the main routine, the DIMENSION must be a constant (6); in the subroutine, the DIMENSION can be a variable (N).

This main routine asks you to input some numbers, and then prints their sum and average, by using STAT2:

```
      DIMENSION X(100)
      PRINT 10
10    FORMAT (1X,'HOW MANY NUMBERS WOULD YOU LIKE TO GIVE ME?')
      READ *, N
      PRINT 20
20    FORMAT (1X,'TYPE THE NUMBERS')
      READ *, (X(I), I=1,N)
      CALL STAT(X, N, SUM, AVERAG)
      PRINT 10, SUM,AVERAG
10    FORMAT (1X,'THE SUM IS',G14.6,' AND THE AVERAGE IS',G14.6)
      END
```

That DIMENSION statement lets you input up to 100 numbers.

If a main routine says DIMENSION X(100), and X is passed between the main routine and the subroutine, the subroutine's DIMENSION statement must say no more than X(100): if the subroutine's DIMENSION statement says X(N), the N must be no more than 100.

For *double* subscripts, the main routine's DIMENSION statement must be *exactly* the same as the subroutine's. For example, if the main routine says DIMENSION X(25,4), the subroutine must say DIMENSION X(25,4) also: it must not say DIMENSION X(20,3); if it says DIMENSION X(M,N), the M must be exactly 25.

The *Scientific Subroutine Package (SSP)* is a collection of subroutines written by IBM that do statistics, calculus, equation-solving, and advanced math. The *Calcomp subroutines* make the computer operate a Calcomp plotter (a device that draws fancy shapes on paper, by using a felt-tip or ballpoint pen). Most large computers store the SSP and Calcomp subroutines on disk permanently. Ask the people in your computer center how to combine your own main routines with those subroutines, and how to put your *own* library of subroutines onto the disk.

If you and your friends want to write a big program together, divide the problem into a main routine and several subroutines, and have each person write one routine. Although you'll need a group conference to decide what the SUBROUTINE and CALL statements will be, you don't have to agree on the names of variables, since the names of subroutine variables have nothing to do with the names of main-routine variables.

YOUR OWN FUNCTIONS

Instead of using functions such as SQRT and ABS, you can invent your *own* functions. Here's how to invent a function called F, and how to use the function you've invented:

```
      A=F(5.2)+1.0
      PRINT 10, A
10    FORMAT (1X,G14.6)
      END

      FUNCTION F(X)
      F=2.0*X
      RETURN
      END
```

The program consists of two routines: the first routine is the *main* routine; the second routine is the definition of the FUNCTION F. Each routine ends with the word END. The main routine's top line requires the computer to find F(5.2), so the computer hunts for the definition of FUNCTION F. The definition say F is twice 5.2, which is 10.4. So A is 11.4. The computer will print:

```
 11.40000
```

Like a subroutine, a function definition can be very long and have many variables in parentheses. The function's name can be F or G or any other name, such as MASS. If the function's name begins with I, J, K, L, M, or N, the computer will assume its value is an integer. If you want the function MASS(X) to be double precision instead of an integer, begin your main routine by saying—

```
      DOUBLE PRECISION MASS
```

and begin your function definition with this line:

```
      DOUBLE PRECISION FUNCTION MASS(X)
```

The computer can write its answers onto a data file:

```
      A=14.6+75.2
      WRITE(3,10) A
10    FORMAT (1X,G14.6)
      WRITE(7,20)
20    FORMAT (1X,'TINA, PLEASE TICKLE MY TUBA')
      END
```

Since A is 89.8, the second line makes the computer write 89.8 onto "file 3", using line 10's format. The next pair of lines make the computer write onto "file 7", using line 20's format; so the computer will write TINA, PLEASE TICKLE MY TUBA onto file 7.

Before running that program, tell the computer where to put files 3 and 7. You can make the computer put them on a disk, line printer, tape, terminal, or wherever else you please. Ask the people in your computer center how to tell the computer where to put the files.

If your program has many statements saying to write onto file 3, the computer will write many lines onto file 3, so the file will become long.

Use carriage controls only if the file is on a line printer or terminal.

If you say—

```
READ(4,30) X
```

the computer will read the value of X from file 4, using line 30's format. If you say file 4 is the card reader, the computer will wait for you to feed in a card; if you say file 4 is your terminal, the computer will wait for you to type on the terminal; if you say file 4 is on a tape or disk, the computer will assme you created the file before running the program.

For handling files, the word READ works even on PDP computers.

For PDP-10 and PDP-20 computers, here's how to make file 3 be on disk and named JOE...Near the beginning of your program, say:

```
OPEN(UNIT=3, DEVICE='DSK', FILE='JOE.')
```

Near the end of your program, say:

```
CLOSE(UNIT=3)
```

CHAPTER 5

PASCAL

FUN

WHAT'S PASCAL?

Imagine that you're running for President, against Calvin Coolidge. To win, you must pass Cal in the polls.

You must pass Cal. "Pass Cal" is the correct way to pronounce "PASCAL", the name of the computer language that programmers are falling in love with.

Of the four popular languages— FORTRAN, COBOL, BASIC, and PASCAL— the newest, most modern, and most elegant is PASCAL. Many programmers who've used FORTRAN are switching to PASCAL. On microcomputers, the two most popular languages are BASIC and PASCAL. Many professors feel PASCAL does a better job of training a student's mind than BASIC, FORTRAN, or COBOL.

Unfortunately, PASCAL is harder to learn than BASIC. But if you learn PASCAL and write your programs in PASCAL, your programs will become more neatly organized, and will therefore contain fewer hidden errors than if written in BASIC.

The original version of PASCAL was invented by Niklaus Wirth in Switzerland, for a CDC maxicomputer. Later, versions of PASCAL were developed for minicomputers and microcomputers. The most popular microcomputer version was invented at the University of California at San Diego (UCSD), and is called *UCSD PASCAL*. Another version is called *ANSI standard PASCAL*. In this chapter, I'll steer a middle course between those versions.

SIMPLE PROGRAMS

Here's a simple PASCAL program, called WINE:

```
PROGRAM WINE;
BEGIN;
WRITELN('I WOULD LIKE TO KISS');
WRITELN('YOUR BOTTLE OF WINE');
END.
```

The top line says the program is called WINE. The guts of the program begins with the word BEGIN, and ends with the word END. Every line ends with a semicolon, except that the bottom line ends with a period.

When you run the program, the computer will write this on your terminal:

```
I WOULD LIKE TO KISS
YOUR BOTTLE OF WINE
```

Weird computers

On CDC computers, the top line should say INPUT and OUTPUT, like this:

```
PROGRAM WINE(INPUT,OUTPUT);
```

On PDP-10 and PDP-20 computers, each WRITELN statement should say TTY, like this:

```
WRITELN(TTY,'I WOULD LIKE TO KISS');
WRITELN(TTY,'YOUR BOTTLE OF WINE');
```

MULTIPLE WRITING

This program belongs in your bathroom:

```
PROGRAM SMELLY;
BEGIN;
WRITELN('FAR','TIN','G');
END.
```

The computer will write FAR, TIN, and G all on the same line, like this:

```
FARTING
```

On PDP-10 and PDP-20 computers, the WRITELN must say TTY, like this:

```
WRITELN(TTY,'FAR','TIN','G');
```

MATH

Like FORTRAN, PASCAL uses integers and reals.

An integer has no decimal point. Here are some integers:

```
723          0          -4
```

A real has a decimal point; for example, -28.3 is a real. You must put digits before the decimal point, and digits after it:

Correct reals	Incorrect
0.37	.37
5.0	5.
6.0E24	6E24

In that table, the entries marked "incorrect" are incorrect in PASCAL, even though they're okay in FORTRAN.

Like FORTRAN, PASCAL uses these four operators:

```
+     -     *     /
```

If you combine a real with a real, the answer is real. If you combine a real with an integer, the answer is still real. If you combine an integer with an integer, the answer is an integer, *except for division*:

7+4 is the integer 11
7-4 is the integer 3
7*4 is the integer 28
7/4 is the real 1.75 (In FORTRAN, 7/4 would be the integer 1 instead.)

This program makes the computer do all that arithmetic:

```
PROGRAM MATH;
BEGIN;
WRITELN(7+4);
WRITELN(7-4);
WRITELN(7*4);
WRITELN(7/4);
END.
```

Here's what the computer will write; the symbol ▮ represents a blank space:

UCSD PASCAL

```
11
3
28
▮1.75
```

UCSD PASCAL puts a blank space before every *real* number that's not negative.

CDC PASCAL

```
▮▮▮▮▮▮▮▮11
▮▮▮▮▮▮▮▮▮3
▮▮▮▮▮▮▮▮28
▮+1.7500000000000E+000
```

To write an integer, CDC PASCAL writes enough blanks to make 10 characters altogether. To write a real, CDC PASCAL uses E notation: it writes a blank space before the number, 13 digits after the decimal point, and a 3-digit exponent.

Let's try writing all those numbers on the same line:

```
PROGRAM LINE;
BEGIN;
WRITELN(7+4,7-4,7*4,7/4);
END.
```

Here's what the computer will write:

UCSD PASCAL

```
11328▮1.75
```

CDC PASCAL

```
▮▮▮▮▮▮▮▮11▮▮▮▮▮▮▮▮▮3▮▮▮▮▮▮▮▮28▮+1.7500000000000000E+000
```

PASCAL lets you use these functions:

Function	Meaning
ABS(-6)	the absolute value of -6; it's 6
SQR(3)	the square of 3; it's 9
SQRT(9)	the square root of 9; it's 3
SIN(2)	the sine of 2 radians
COS(2)	the cosine of 2 radians
ARCTAN(2)	the arctangent of 2, in radians
EXP(5)	e^5, where e is 2.71828182845904523536
LN(9)	the natural logarithm of 9; it's $\log_e 9$

The functions SQRT, SIN, COS, ARCTAN, EXP, and LN are called "transcendental". To use transcendental functions on an Apple, you must insert this line immediately under the line that says PROGRAM:

```
USES TRANSCEND;
```

The number in parentheses can be either an integer or a real. The computer's answer is usually a real (exception: for ABS and SQR, the computer's answer is an integer if the number in parentheses is an integer).

For example, this program computes the square root of 9:

```
PROGRAM ROOT;
BEGIN;
WRITELN(SQRT(9));
END.
```

The computer will write:

UCSD PASCAL

```
▮3.0
```

CDC PASCAL

```
▮+3.0000000000000E+000
```

Don't confuse SQR with SQRT. The SQR means *square*; the SQRT means square *root*. To find the square of 3, say SQR(3), or say 3*3. In BASIC, you could say 3↑2; in FORTRAN, you could say 3**2; but in PASCAL, you can't use the symbols ↑ or **; use SQR instead.

To turn a real number into an integer, use these functions:

Function	Meaning
ROUND(3.9)	3.9 rounded to the nearest integer; it's 4
TRUNC(3.9)	3.9 truncated (by removing the decimal point); it's 3

VARIABLES

You can use variables:

```
PROGRAM VARY;                       The program's name is VARY.
VAR                                 Here are the variables....
        FRED,MARTHA: INTEGER;               FRED and MARTHA are integers;
        JILL,TOM: REAL;                     JILL and TOM are reals.
BEGIN;                              Now let's begin the program....
FRED:=2;                            FRED (which is an integer) is 2.
MARTHA:=4+5;                        MARTHA (which is an integer) is 9.
JILL:=9.2;                          JILL (which is a real) is 9.2.
TOM:=1.4+2.3;                       TOM (which is a real) is 3.7.
WRITELN(FRED*MARTHA,JILL+TOM); The computer will write 18 and 12.9.
END.
```

The second line says VAR. The word VAR doesn't have a semicolon after it; the lines underneath VAR are indented. The indentation is optional, but is a good habit. To indent, hit the space bar several times, or type a controlled I. In those indented lines, you must list all the program's variables. You must say which variables are integers and which are reals.

In PASCAL, a variable's name can be as long as you like: the name can be FRED or WASHINGTON or even SUPERCALIFRAGILISTICEXPIALIDOCIOUS. The name can consist of letters and digits, but must begin with a letter. The computer looks at just the name's first 8 characters; so if you say—

```
SUPERCALIFRAGILISTICEXPIALIDOCIOUS:=25000;
```

the computer will think you said:

```
SUPERCAL:=25000;
```

In PASCAL, you must put a colon in front of the equal sign:

Correct	Incorrect
FRED:=2;	FRED=2;

You can't set an integer variable equal to a real. For example, if you say—

```
VAR
        ANN: REAL;
        BILL: INTEGER;
```

you can*not* say:

```
ANN:=3.9;
BILL:=ANN;
```

Instead, you must say—

```
BILL:=ROUND(ANN);
```

or

```
BILL:=TRUNC(ANN);
```

READ

This program works in UCSD PASCAL:

```
PROGRAM MYSQRT;
VAR
        X: REAL;
BEGIN;
WRITELN('WHAT IS YOUR FAVORITE NUMBER?');
READ(X);
WRITELN('ITS SQUARE ROOT IS',SQRT(X));
END.
```

When you run the program, the computer asks:

```
WHAT IS YOUR FAVORITE NUMBER?
```

Then the READ(X) statement makes the computer wait for you to input a value of X. The computer expects you to input a real number; if you input an integer instead, the computer will automatically turn it into a real. For example, if you input 9, the computer will automatically turn it into 9.0, and will write:

```
ITS SQUARE ROOT IS 3.0
```

Weird computers

CDC, PDP-10, and PDP-20 computers ignore the first line you input. So you must input a blank line, and *afterwards* input the number you want to take the square root of.

On PDP-10 and PDP-20 computers, if you say WRITELN and then READ, you must insert the word BREAK between them:

```
WRITELN('WHAT IS YOUR FAVORITE NUMBER?');
BREAK;
READ(X);
```

IF

The computer can criticize your age:

```
PROGRAM KIDDO;
VAR
        AGE: INTEGER;
BEGIN;
WRITELN('HOW OLD ARE YOU?');
READ(AGE);
IF AGE<18 THEN
        BEGIN;
        WRITELN('YOU ARE STILL A MINOR.');
        WRITELN('AH, THE JOYS OF YOUTH!');
        END;
WRITELN('GLAD TO MEET YOU.');
END.
```

The computer asks:

```
HOW OLD ARE YOU?
```

If you input a number less than 18, the computer replies:

```
YOU ARE STILL A MINOR.
AH, THE JOYS OF YOUTH!
GLAD TO MEET YOU.
```

If you input a number that's at least 18, the computer says just this:

```
GLAD TO MEET YOU.
```

In that program the line that says "IF" is a heading: the lines underneath it are indented. Because the "IF" line is a heading, do *not* put a semicolon at the end of that line!

To make the program fancier, insert extra lines:

```
PROGRAM MINOR;
VAR
        AGE: INTEGER;
BEGIN;
WRITELN('HOW OLD ARE YOU?');
READ(AGE);
IF AGE<18 THEN
        BEGIN;
        WRITELN('YOU ARE STILL A MINOR.');
        WRITELN('AH, THE JOYS OF YOUTH!');
→→      END
→→ELSE
→→      BEGIN;
→→      WRITELN('I AM GLAD TO HEAR YOU ARE AN ADULT!);
→→      WRITELN('NOW WE CAN HAVE SOME ADULT FUN!');
→→      END;
  WRITELN('GLAD TO MEET YOU.');
  END.
```

If your age is less than 18, the computer will say:

```
YOU ARE STILL A MINOR.
AH, THE JOYS OF YOUTH!
GLAD TO MEET YOU.
```

If your age is *not* less than 18, the computer will say:

```
I AM GLAD TO HEAR YOU ARE AN ADULT.
NOW WE CAN HAVE SOME ADULT FUN!
GLAD TO MEET YOU.
```

In that program, the line that says "ELSE" is a heading, so do *not* put a semicolon after the "ELSE". Immediately above the word ELSE, you'll see the word END; do *not* put a semicolon after that END.

Like BASIC, PASCAL uses these symbols in the IF line:

```
<       >       =       <=      >=      <>      AND     OR
```

In the IF line, the symbol for "equals" is "="; outside the IF line, the symbol for "equals" is ":=".

Instead of saying—

```
IF I=3 OR I=8 OR I=25 OR I=95 THEN
```

you can say:

```
IF I IN [3,8,25,95] THEN
```

LOOPS

FOR

This program makes the computer get drunk:

```
PROGRAM BEER;
VAR
        I: INTEGER;
BEGIN;
FOR I := 6 TO 10 DO
        BEGIN;
        WRITELN('I DRANK ',I,' BOTTLES OF BEER');
        WRITELN('HOORAY!');
        END;
WRITELN('NOW I AM DEAD DRUNK');
END.
```

The computer will write:

```
I DRANK 6 BOTTLES OF BEER
HOORAY!
I DRANK 7 BOTTLES OF BEER
HOORAY!
I DRANK 8 BOTTLES OF BEER
HOORAY!
I DRANK 9 BOTTLES OF BEER
HOORAY!
I DRANK 10 BOTTLES OF BEER
HOORAY!
NOW I AM DEAD DRUNK
```

In that program, the line that says "FOR" is a heading, so do *not* put a semicolon at the end of that line.

If you want the computer to count backwards— from 10 down to 6— change the FOR line to this:

```
FOR I := 10 DOWNTO 6 DO
```

REPEAT...UNTIL

This program plays a guessing game:

```
PROGRAM GAME;
VAR
        GUESS: INTEGER;
BEGIN;
WRITELN('I AM THINKING OF A NUMBER FROM 1 TO 10');
REPEAT
        WRITELN('YOU HAVE NOT GUESSED MY NUMBER YET');
        WRITELN('WHAT IS MY NUMBER?');
        READ(GUESS);
UNTIL GUESS=6;
WRITELN('CONGRATULATIONS! YOU GUESSED IT! MY NUMBER IS 6');
END.
```

The computer begins the game by saying:

```
I AM THINKING OF A NUMBER FROM 1 TO 10
```

Then the computer repeats the following procedure: it says YOU HAVE NOT GUESSED MY NUMBER YET, asks WHAT IS MY NUMBER, waits for the human to input a guess, and checks whether the human guessed the number 6. It repeats the procedure again and again, until the human finally guesses the number 6. Then the computer says:

```
CONGRATULATIONS! YOU GUESSED IT! MY NUMBER IS 6
```

In that program, the computer repeats the indented lines, until GUESS is 6.

If a program contains the word REPEAT, it must also contain the word UNTIL. The lines between REPEAT and UNTIL are done repeatedly. After each time, the computer checks whether to repeat again.

Since the REPEAT line is a heading, it has no semicolon.

The UNTIL statement resembles an IF statement; it can use these symbols:

```
<       >       =       <=      >=      <>      AND     OR      IN
```

Here's another example. Let's make the computer start with the number 3, and keep doubling it, like this:

```
3
6
12
24
48
96
etc.
```

Let's make the computer keep doubling, but not go over 1000. So altogether, let's make the computer write:

```
3
6
12
24
48
96
192
384
768
THOSE ARE ALL THE NUMBERS BELOW 1000
```

Here's the program:

```
PROGRAM DOUBLE;
VAR
        I: INTEGER;
BEGIN;
I:=3;                   Start I at 3.
REPEAT                  Do the indented lines repeatedly, until I>=1000.
        WRITELN(I);
        I:=I*2;
UNTIL I>=1000;
WRITELN('THOSE ARE ALL THE NUMBERS LESS THAN 1000');
END.
```

WHILE

The word WHILE resembles the word REPEAT. For example, the previous program (which doubled 3 repeatedly) can be rewritten, by using WHILE:

```
  PROGRAM DOUBLE;
  VAR
        I: INTEGER;
  BEGIN;
  I:=3;                 Start I at 3.
→→WHILE I<1000 DO       If I<1000, do the indented lines; and repeat the lines
→→        BEGIN;        again and again, as long as I remains less than 1000.
          WRITELN(I);
          I:=I*2;
→→        END;
  WRITELN('THOSE ARE ALL THE NUMBERS LESS THAN 1000');
  END.
```

The indented lines say to WRITELN(I) and then to double I. The computer will repeat those indented lines many times. Before each repetition, the computer checks to make sure I is still less than 1000. When I passes 1000, the computer stops looping, and instead proceeds to the next idea, which says to write THOSE ARE ALL THE NUMBERS LESS THAN 1000.

Notice that these two structures serve almost the same purpose:

Using REPEAT...UNTIL	Using WHILE
REPEAT	WHILE I<1000 DO
etc.	BEGIN;
UNTIL I>=1000;	*etc.*
	END;

Each of those structures makes the computer repeat the indented lines, and makes the computer check whether I is still less than 1000. If you say REPEAT...UNTIL, the check is done at the bottom of the loop, so the computer goes through the loop once before checking. If you say WHILE, the check is done at the top of the loop, so the computer checks whether I<1000 before doing the loop the first time.

EXOTIC FEATURES

ALTERNATIVES

Every statement must end with a semicolon. You can put several statements on the same line:

Normal
```
A:=3;
B:=7.6;
WRITELN(A+B);
```

Alternative
```
A:=3; B:=7.6; WRITELN(A+B);
```

You can continue a statement onto the next line:

Normal
```
A:=5-2+1;
```

Alternative
```
A:=
5-2
+1;
```

But do *not* divide a statement in the middle of a word, number, symbol, or string:

Normal
```
IF AGE>=18 THEN
```

Okay
```
IF AGE
>=18 THEN
```

Wrong, because middle of word
```
IF AG
E>=18 THEN
```

Wrong, because middle of number
```
IF AGE>=1
8 THEN
```

Wrong, because middle of symbol
```
IF AGE>
=18 THEN
```

You can omit the semicolon after BEGIN:

Normal
```
BEGIN;
```

Alternative
```
BEGIN
```

You can omit a semicolon, if the next word is END:

Normal
```
A:=3;
END;
```

Alternative
```
A:=3
END;
```

You can omit a semicolon, if the next word is UNTIL:

Normal
```
        I=I*2;
UNTIL I>=1000;
```

Alternative
```
          I=I*2
UNTIL I>=1000;
```

Here's a short cut that's more useful: underneath a heading, if you see *only one* indented line (besides BEGIN and END), move that indented line, and put it next to the heading (and omit the BEGIN and END).

Long way

```
FOR I := 1 TO 10 DO
        BEGIN;
        WRITELN(I);
        END;

WHILE A<100 DO
        BEGIN;
        A:=A+7;
        END;

IF AGE>=65 THEN
        BEGIN;
        WRITELN('MUST RETIRE');
        END;

IF WEIGHT>=200 THEN
        BEGIN;
        WRITELN('FAT');
        END
ELSE
        BEGIN;
        WRITELN('OKAY');
        END;

REPEAT
        A:=A+7;
UNTIL A>100;
```

Short cut

```
FOR I := 1 TO 10 DO WRITELN(I);

WHILE A<100 DO A:=A+7;

IF AGE>=65 THEN WRITELN('MUST RETIRE');

IF WEIGHT>=200 THEN WRITELN('FAT') ELSE WRITELN('OKAY');

REPEAT A:=A+7 UNTIL A>100;
```

COMMENTS

Here's how to put a comment into your program:

BASIC
```
10 REM  I HATE COMPUTERS
```

FORTRAN
```
C   I HATE COMPUTERS
```

PASCAL
```
{I HATE COMPUTERS}
```

If your terminal's keyboard doesn't have braces, use parentheses and asterisks instead:

PASCAL, if no braces
```
(*I HATE COMPUTERS*)
```

Do *not* put a semicolon after the comment.

GOTO

You can say GOTO.

```
PROGRAM SKIPPER;
LABEL 10;
BEGIN;
WRITELN('MY DOG');
GOTO 10;
WRITELN('NEVER');
10: WRITELN('DRINKS WHISKEY');
END.
```

The top line says the program's name is SKIPPER. The next line warns the computer that a future line will be labeled "line 10". The rest of the program makes the computer write MY DOG, then skip to line 10, which makes the computer write DRINKS WHISKEY. Altogether, the computer will write:

```
MY DOG
DRINKS WHISKEY
```

Do *not* put a space between GO and TO.

Here's another example:

```
PROGRAM LOVEHATE;
LABEL 10;
BEGIN;
10: WRITELN('LOVE');
WRITELN('HATE');
GOTO 10;
END.
```

The computer will write LOVE and HATE repeatedly:

```
LOVE
HATE
LOVE
HATE
LOVE
HATE
etc.
```

If one line of your program is numbered 10, and another is numbered 20, and another is numbered 100, put this statement at the top of your program:

```
LABEL 10,20,100;
```

Put that statement at the very top of your program— immediately underneath the line that says PROGRAM. The PROGRAM and LABEL statements go above all other lines of your program: they even go above the line that says BEGIN and any line that says VAR.

The largest permissible line number is 9999.

In PASCAL, you'll rarely need to say GOTO; instead, try using the words IF, ELSE, FOR, REPEAT, and WHILE.

Weird computers

In UCSD PASCAL, if you want to say GOTO, you must say (*$G+*) at the top of your program:

```
(*$G+*)
PROGRAM SKIPPER;
LABEL 10;
etc.
```

This program teases you, with the help of some insults:

```
PROGRAM TEASE;

PROCEDURE INSULT;
BEGIN;
WRITELN('YOU ARE STUPID');
WRITELN('YOU ARE UGLY');
END;

(*MAIN ROUTINE*)
BEGIN;
WRITELN('WE ALL KNOW...');
INSULT;
WRITELN('...AND YET WE LOVE YOU');
END.
```

The program is called TEASE. It defines INSULT to be this procedure:

```
WRITELN('YOU ARE STUPID');
WRITELN('YOU ARE UGLY');
```

The main routine makes the computer write 'WE ALL KNOW...', then do the INSULT procedure, then write '...AND YET WE LOVE YOU'; altogether, the computer will write:

```
WE ALL KNOW...
YOU ARE STUPID
YOU ARE UGLY
...AND YET WE LOVE YOU
```

The PROCEDURE, called INSULT, is a subroutine. In PASCAL, subroutines come *before* the main routine. (In BASIC and FORTRAN, subroutines come *after* the main routine, instead.)

The PROCEDURE comes below the program's VAR:

```
PROGRAM LARGER;         The program is called LARGER.
VAR                     The program uses a variable,
        I: INTEGER;     called I, which is an integer.

PROCEDURE DOUBLE;       The subroutine is called DOUBLE.
BEGIN;
I:=2*I;                 The subroutine doubles the value of I,
WRITELN(I);             and writes the new value of I.
END;

(*MAIN ROUTINE*)        Here's the main routine....
BEGIN;
I:=7;                   I starts at 7.
DOUBLE;                 The computer does subroutine DOUBLE, which writes 14.
DOUBLE;                 The computer does subroutine DOUBLE again, which writes 28.
WRITELN(I+1);           The computer will write 29.
END.
```

A variable can stand for a character:

```
PROGRAM RISE;
VAR
        ANN,JOAN: CHAR;       ANN and JOAN will each be a character.
BEGIN;
ANN:='U';                     ANN is the character 'U'.
JOAN:='P';                    JOAN is the character 'P'.
WRITELN(ANN,JOAN);            The computer will write 'U' and 'P'.
END.
```

The computer will write U and P like this:

```
UP
```

You can put a character variable after the word FOR:

```
PROGRAM LOOP;
VAR
        I: CHAR;
BEGIN;
FOR I := 'A' TO 'E' DO WRITELN(I);
END.
```

The computer will write:

```
A
B
C
D
E
```

SUBSCRIPTS

Suppose you want X to be this list of real numbers: 4.2, 71.6, 8.3, 92.6, 403.7, 1.4. Here's how:

```
PROGRAM LIST;
VAR
        X: ARRAY [1..6] OF REAL;
BEGIN;
X[1]:=4.2;
X[2]:=71.6;
X[3]:=8.3;
X[4]:=92.6;
X[5]:=403.7;
X[6]:=1.4;
WRITELN(X[1]+X[2]+X[3]+X[4]+X[5]+X[6]);
END.
```

The computer will write the sum, 581.8.

Subscripts can be negative:

```
PROGRAM MINUS;
VAR
        Y: ARRAY [-2..3] OF REAL;
BEGIN;
Y[-2]:=400.1;
Y[-1]:=274.1;
Y[0]:=9.2;
Y[1]:=8.04;
Y[2]:=0.6;
Y[3]:=-5.0;
WRITELN(Y[-2]+Y[-1]+Y[0]+Y[1]+Y[2]+Y[3]);
END.
```

The computer will write the sum, 687.04.

If you want Z to be a table that has 6 rows and 4 columns of reals, the VAR statement should look like this:

```
VAR
        Z: ARRAY [1..6, 1..4] OF REAL;
```

The number in the 3rd row and 2nd column is called Z[3,2]. The entire first row of Z is called Z[1]; the second row is called Z[2]; etc. For example, if you say—

```
Z[5]:=Z[3];
```

the computer will at the real numbers in the 3rd row of Z, and copy them into the 5th row. Suppose W is another array that has 6 rows and 4 columns; if you say—

```
W:=Z;
```

the computer will look at each real number in Z and copy it into W.

You can have many kinds of arrays: you can have an array of REAL, an array of INTEGER, or even an array of CHAR.

WRITING STRINGS

In UCSD PASCAL, a variable can stand for a string:

```
PROGRAM LOVER;
VAR
        X: STRING;
BEGIN;
X:='I LOVE MY MOTH';
WRITELN(X,'ER');
END.
```

The computer will write 'I LOVE MY MOTH' and then 'ER', like this:

```
I LOVE MY MOTHER
```

Weird computers

PDP computers don't understand STRING; instead of STRING, say PACKED ARRAY:

```
  PROGRAM LOVER;
  VAR
→→        X: PACKED ARRAY [1..14] OF CHAR;   That means: X is a string of 14 characters
  BEGIN;
  X:='I LOVE MY MOTH';
  WRITELN(X,'ER');
  END.
```

(On PDP-10 and PDP-20 computers, omit the word PACKED, and put TTY into the WRITELN statement.)

CDC computers don't understand STRING, and don't know how to write an array; instead of writing the array, you must write each character of the array separately, by using a FOR loop:

```
  PROGRAM LOVER(INPUT,OUTPUT);
  VAR
          X: PACKED ARRAY [1..14] OF CHAR;
→→        I: INTEGER;
  BEGIN;
  X:='I LOVE MY MOTH';
→→FOR I := 1 TO 14 DO WRITE(X[I]);
→→WRITELN('ER');
  END.
```

READING STRINGS

UCSD PASCAL lets the computer read a string:

```
PROGRAM GREET;
VAR
        X: STRING;
BEGIN;
WRITELN('WHAT IS YOUR NAME?');
READ(X);
WRITELN('HELLO ',X,' THE BEAUTIFUL');
END.
```

When you run that program, the computer asks:

```
WHAT IS YOUR NAME?
```

Then the READ(X) statement makes the computer wait for you to input a string X. If you input the name MARILYN MONROE, the computer will write:

```
HELLO MARILYN MONROE THE BEAUTIFUL
```

Weird computers

PDP and CDC computers don't understand STRING; for those computers, you must rewrite the program:

UCSD PASCAL	PDP and CDC, if you know X will have 14 characters	PDP and CDC, if you don't know the length of X
`X: STRING;`	`X: PACKED ARRAY [1..14] OF CHAR;` `I: INTEGER;`	`X: PACKED ARRAY [1..80] OF CHAR;` `I,XLENGTH: INTEGER;`

`READ(X);`	`FOR I := 1 TO 14 DO READ(X[I]);`
	`READLN;`
		`I:=0;`
		`WHILE NOT EOLN DO`
		`  BEGIN;`
		`  I:=I+1;`
		`  READ(X[I]);`
		`  END;`
		`READLN;`
		`XLENGTH:=I;`

`WRITELN('HELLO ',X,`
`' THE BEAUTIFUL');`
	`WRITE('HELLO ');`
	`FOR I := 1 TO 14 DO WRITE(X[I]);`
	`WRITELN(' THE BEAUTIFUL');`
		`WRITE('HELLO ');`
		`FOR I := 1 TO XLENGTH DO`
		`  WRITE(X[I]);`
		`WRITELN(' THE BEAUTIFUL');`

On PDP-10 and PDP-20 computers, omit the word PACKED, and say (TTY) for READ and WRITE and READLN and WRITELN and EOLN.

CHAPTER 6

COBOL

FUN

WHAT'S COBOL?

Like BASIC, FORTRAN, and PASCAL, *COBOL* is a computer language. "COBOL" is pronounced "koe ball" and stands for "COmmon Business Oriented Language".

COBOL solves *business* problems that involve *large* files of data. Therefore, COBOL is most popular among *business*men who have *maxi*computers. But today, you can use COBOL even on minicomputers and microcomputers.

In the "help wanted" section of your local newspaper, most ads that say "programmer wanted" are placed by businessmen that have maxicomputers. Therefore, most of the "programmer wanted" ads demand a knowledge of COBOL. So if you want to get a job through the "help wanted" section, a knowledge of COBOL will help you more than BASIC, FORTRAN, or PASCAL.

The first version of COBOL was called *COBOL 60*, because it was invented in 1960. Then came *COBOL 61*, *COBOL 65*, *COBOL 68*, and *COBOL 74*. Today, most minicomputers and microcomputers use a slight variation of COBOL 74; most maxicomputers use a compromise between COBOL 68 and COBOL 74. In this chapter, you'll learn how to write COBOL programs on today's popular computers.

During the 1960's and early 1970's, COBOL programmers used a style called "easy programming". Today, most COBOL programmers use a more sophisticated style, called "*structured* programming". This chapter explains structured programming. Though it's harder to learn than easy programming, it will make your boss kiss you.

Every COBOL program is written as an outline. Here's a short outline; to turn it into a COBOL program, just fill in the blanks:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        The program's name.
AUTHOR.
        Your name.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.

DATA DIVISION.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        What you want the computer to do.
        STOP RUN.
```

For example, here's a COBOL program I wrote:

Program	Reason
`IDENTIFICATION DIVISION.`	
`PROGRAM-ID.`	
`        HARRY.`	The program's name is HARRY.
`AUTHOR.`	
`        RUSS WALTER.`	My name is Russ Walter.
`ENVIRONMENT DIVISION.`	
`CONFIGURATION SECTION.`	
`SOURCE-COMPUTER.`	
`        DECSYSTEM-20.`	My computer is a DECsystem-20 (which is a PDP-20).
`OBJECT-COMPUTER.`	
`        DECSYSTEM-20.`	
`DATA DIVISION.`	
`PROCEDURE DIVISION.`	
`MAIN-ROUTINE.`	
`        DISPLAY "LIFE STINKS".`	I want the computer to print "LIFE STINKS".
`        STOP RUN.`	

When I run that program, the computer will print:

```
LIFE STINKS
```

Every COBOL program consists of four parts. The first part of the program is called the IDENTIFICATION DIVISION: it includes the program's name and the programmer's name. The second part of the program is called the ENVIRONMENT DIVISION: it includes the computer's name. The third part of the program is called the DATA DIVISION; for a simple program, the DATA DIVISION is blank. The fourth part of the program is the PROCEDURE DIVISION: it says what you want the computer to do.

The order is important: the IDENTIFICATION DIVISION must come *first*, then the ENVIRONMENT DIVISION, then the DATA DIVISION, and finally the PROCEDURE DIVISION. So to become an expert COBOL programmer, you must memorize: "IDENTIFICATION, ENVIRONMENT, DATA, PROCEDURE".

To memorize that easily, memorize this easy sentence: "I enjoy data processing". In that sentence, the words begin with the letters "I E D P"— and so do the four COBOL divisions.

In the program, each blank that you fill is called a *paragraph*. In the first paragraph, write the program's name; in the next paragraph, write your own name; in the next two paragraphs, write the computer's name; and in the last paragraph, write what you want the computer to do. The first paragraph is called the PROGRAM-ID; the next paragraph is called the AUTHOR; the next two paragraphs are called the SOURCE-COMPUTER and the OBJECT-COMPUTER; and the last paragraph is called the MAIN-ROUTINE.

Each paragraph is indented. To indent on PDP and Eclipse computers, type a controlled I.

In COBOL, the only important punctuation mark is the period. When writing a simple program, put a period at the end of each line. COBOL never requires commas or semicolons.

Don't forget the hyphens! Put a hyphen in PROGRAM-ID, SOURCE-COMPUTER, OBJECT-COMPUTER, and MAIN-ROUTINE.

Use correct spacing:

Right	Wrong	Wrong	Wrong
`DISPLAY "BURP".`	`DIS PLAY "BURP".`	`DISPLAY"BURP".`	`DISPLAY "BURP" .`

In the paragraphs that are called SOURCE-COMPUTER and OBJECT-COMPUTER, you must type the computer's name correctly. Here are the correct names for some famous computers:

```
IBM-360.
IBM-370.
PDP-11.
DECSYSTEM-10.    (It means you have a PDP-10.)
DECSYSTEM-20.    (It means you have a PDP-20.)
ECLIPSE C300.
6600.            (It means you have a CDC 6600.)
```

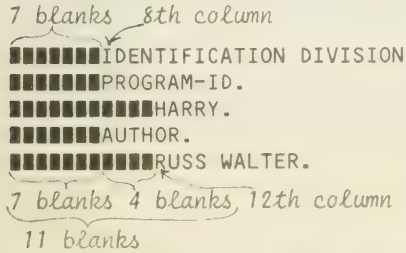
Don't forget the hyphens!

Old-fashioned computers

On IBM computers, instead of using quotation marks, you must use apostrophes:

Most computers	IBM computers
`DISPLAY "LIFE STINKS".`	`DISPLAY 'LIFE STINKS'.`

On IBM and CDC computers, you must indent the entire program, like this (each ∎ represents a blank space):



On those computers, each paragraph begins in column 12, and must not go farther to the right than column 72. (The computer ignores everything in columns 73-80.)

Try it

Go ahead: try writing your *own* COBOL program. In the PROCEDURE DIVISION, remember to say DISPLAY:

BASIC
```
10 PRINT "LIFE STINKS"
```

FORTRAN
```
      PRINT 10
10    FORMAT(1X,'LIFE STINKS')
```

PASCAL
```
WRITELN('LIFE STINKS');
```

COBOL
```
DISPLAY "LIFE STINKS".
```

ABRIDGMENTS

If you're lazy, you can omit the AUTHOR paragraph:

Complete IDENTIFICATION DIVISION
```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        HARRY.
AUTHOR.
        RUSS WALTER.
```

Abridged IDENTIFICATION DIVISION
```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        HARRY.
```

If you're lazy, *and you're using a PDP-10, PDP-20, Eclipse, or IBM computer*, you can omit the CONFIGURATION SECTION, SOURCE-COMPUTER, and OBJECT-COMPUTER:

Complete ENVIRONMENT DIVISION
```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name.
```

Abridged ENVIRONMENT DIVISION
```
ENVIRONMENT DIVISION.
```

If you're *very* lazy, and you're using a PDP-10 or PDP-20 computer, you can abridge the program even further, so that the entire program looks like this:

```
IDENTIFICATION DIVISION.
PROCEDURE DIVISION.
        DISPLAY "LIFE STINKS".
        STOP RUN.
```

But if you're working for a big company, your employer will expect you to *not* be lazy: if you're lazy, you get fired!

FANCY DISPLAYS

You've seen that every COBOL program consists of four divisions: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. The most important division is the PROCEDURE DIVISION. Let's look at it more closely.

Here's a cute PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "BILLIE AND BONNIE".
        DISPLAY "BURP".
        STOP RUN.
```

It makes the computer display:

```
BILLIE AND BONNIE
BURP
```

Another example:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "FLU" "SHED".
        STOP RUN.
```

The computer will display FLU and SHED on the same line:

```
FLUSHED
```

PERFORM

Let's make the computer display "I LOVE YOU", then display "I HATE YOU" six times, then display "I AM CONFUSED", like this:

```
I LOVE YOU
I HATE YOU
I HATE YOU
I HATE YOU
I HATE YOU
I HATE YOU
I HATE YOU
I AM CONFUSED
```

Here's the PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "I LOVE YOU".
        PERFORM EXPRESS-THE-HATRED 6 TIMES.
        DISPLAY "I AM CONFUSED.
        STOP RUN.
EXPRESS-THE-HATRED.
        DISPLAY "I HATE YOU".
```

The PROCEDURE DIVISION consists of two paragraphs: the first paragraph is called the MAIN-ROUTINE; the second paragraph is called EXPRESS-THE-HATRED.

The computer obeys the first paragraph: it displays "I LOVE YOU", then performs the EXPRESS-THE-HATRED 6 times, then displays "I AM CONFUSED", and finally stops.

When you invent your own PROCEDURE DIVISION, the first paragraph should be called the MAIN-ROUTINE; for the other paragraphs underneath, invent whatever names you like (such as EXPRESS-THE-HATRED). A paragraph's name should be hyphenated, and should contain no more than 30 characters. (EXPRESS-THE-HATRED contains 18 characters, so it's okay.) The first paragraph is the main routine, the paragraphs underneath are subroutines. The bottom line of the main routine should say STOP RUN. In the middle of the main routine, you should say to PERFORM the subroutines.

In the example above, the main routine says to PERFORM the EXPRESS-THE-HATRED subroutine 6 times. If you'd like to see more hatred, say 100 times instead of 6:

```
        PERFORM EXPRESS-THE-HATRED 100 TIMES.
```

If you'd rather see just a *little* hatred, say just—

```
        PERFORM EXPRESS-THE-HATRED 1 TIMES.
```

or say just:

```
        PERFORM EXPRESS-THE-HATRED.
```

Let's make the computer display:

```
I KNOW THAT
YOU ARE DRIVING
ME CRAZY
YOU ARE DRIVING
ME CRAZY
YOU ARE DRIVING
ME CRAZY
YOU ARE DRIVING
ME CRAZY
YOU ARE DRIVING
ME CRAZY
AND YET I LOVE YOU
```

Here's the PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "I KNOW THAT".
        PERFORM ACT-AS-IF-GOING-INSANE 5 TIMES.
        DISPLAY "AND YET I LOVE YOU".
        STOP RUN.
ACT-AS-IF-GOING-INSANE.
        DISPLAY "YOU ARE DRIVING".
        DISPLAY "ME CRAZY".
```

Let's make the computer display:

```
THE ASTRONAUTS GO
UP
UP
UP
UP
UP
UP
UP
AND THEN THEY COME
DOWN
DOWN
DOWN
DOWN
DOWN
DOWN
DOWN
```

Here's the PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "THE ASTRONAUTS GO".
        PERFORM SHOW-THE-ASTRONAUTS-RISING 7 TIMES.
        DISPLAY "AND THEN THEY COME".
        PERFORM SHOW-THE-ASTRONAUTS-FALLING 7 TIMES.
        STOP RUN.
SHOW-THE-ASTRONAUTS-RISING.
        DISPLAY "UP".
SHOW-THE-ASTRONAUTS-FALLING.
        DISPLAY "DOWN".
```

Let's make the computer display:

```
YOU ARE VERY SWEET
HA-HA-HA!
HO-HO-HO!
YOU CANNOT BE BEAT
HA-HA-HA!
HO-HO-HO!
YOUR LIPS ARE LIKE WINE
HA-HA-HA!
HO-HO-HO!
BUT YOU SMELL LIKE TURPENTINE
HA-HA-HA!
HO-HO-HO!
YOU STINK!
```

Here's the PROCEDURE DIVISION:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "YOU ARE VERY SWEET".
        PERFORM LAUGH-A-LOT.
        DISPLAY "YOU CANNOT BE BEAT".
        PERFORM LAUGH-A-LOT.
        DISPLAY "YOUR LIPS ARE LIKE WINE".
        PERFORM LAUGH-A-LOT.
        DISPLAY "BUT YOU SMELL LIKE TURPENTINE".
        PERFORM LAUGH-A-LOT.
        DISPLAY "YOU STINK!".
        STOP RUN.
LAUGH-A-LOT.
        DISPLAY "HA-HA-HA!".
        DISPLAY "HO-HO-HO!".
```

VARIABLES

STRING VARIABLES

Like BASIC, FORTRAN, and PASCAL, COBOL lets you use variables. The name of a variable can be a letter (like X or Y) or a hyphenated phrase (like NUMBER-OF-BULLIES-I-SQUIRTED). A hyphenated phrase can have up to 30 characters.

To use a variable, you must describe it in the data division, as in this example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        JUNK.
AUTHOR.
        RUSS WALTER.          Instead of "RUSS WALTER", write your own name.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        DECSYSTEM-20.         Instead of "DECSYSTEM-20", write your own computer's name.
OBJECT-COMPUTER.
        DECSYSTEM-20.

DATA DIVISION.
WORKING-STORAGE SECTION.      In the DATA DIVISION, you must say WORKING-STORAGE SECTION
01      K PICTURE IS XXX.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE "HER" TO K.
        DISPLAY "PUS" K "S".
        STOP RUN.
```

In the DATA DIVISION's WORKING-STORAGE SECTION, the "01 K" says K is a variable. The PICTURE IS XXX says K is a string that has three characters; each X stands for a character. In the PROCEDURE DIVISION, the first sentence makes K become this 3-character string: "HER". The next sentence makes the computer display:

```
PUSHERS
```

When you type that program, make sure you put a hyphen between WORKING and STORAGE. If you forget the hyphen, the computer will act crazy, and will say that your program contains many, many errors.

Suppose you change the picture from XXX to XX, so that the DATA DIVISION and PROCEDURE DIVISION look like this:

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
→→01      K PICTURE IS XX.

  PROCEDURE DIVISION.
  MAIN-ROUTINE.
          MOVE "HER" TO K.
          DISPLAY "PUS" K "S".
          STOP RUN.
```

K will be a string having only two characters. When the computer tries to move "HER" to K, only the first two characters of "HER" will fit, so K will be "HE". The computer will display:

```
PUSHES
```

Suppose you change the picture to XXXX. K will have four characters. When the computer tries to move "HER" to K, it needs to move a fourth character also, so it moves a blank space at the end, which makes K be "HER ". The computer will display:

```
PUSHER S
```

This program shows how revolutionary politics leads to revolutionary clothing:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      FIRST-PRESIDENT PICTURE IS XXXXXXXXXX.
01      CLEANING-METHOD PICTURE IS XXXX.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE "WASHINGTON" TO FIRST-PRESIDENT.
        MOVE FIRST-PRESIDENT TO CLEANING-METHOD.
        DISPLAY CLEANING-METHOD " MY BLUE JEANS".
        STOP RUN.
```

Above the DATA DIVISION, write your own IDENTIFICATION DIVISION and ENVIRONMENT DIVISION. The DATA DIVISION says FIRST-PRESIDENT will be a string having ten characters, and CLEANING-METHOD will be a string having four. The first sentence of the MAIN-ROUTINE makes FIRST-PRESIDENT be "WASHINGTON". The next sentence tries to move "WASHINGTON" to CLEANING-METHOD; but because of CLEANING-METHOD's picture, the computer moves "WASH" instead. The computer will display:

```
WASH MY BLUE JEANS
```

To make sure you understand the word MOVE, examine this example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      C PICTURE IS XXX.
01      D PICTURE IS XXX.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE "CAT" TO C.      C becomes "CAT".
        MOVE C TO D.          D becomes "CAT".
        DISPLAY C.            Since C is still "CAT", the computer displays "CAT".
        MOVE "HE" TO C.       C becomes "HE ".
        DISPLAY C "BLED".     The computer displays "HE BLED".
        STOP RUN.
```

COBOL allows abbreviations. You can say PIC instead of PICTURE IS, and X(7) instead of XXXXXXX.

Let's make the computer add 53 and 4, and display the sum, 57. Here's how:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC 99.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        COMPUTE K = 53 + 4.
        DISPLAY K.
        STOP RUN.
```

The PIC 99 says K is a number having two digits. (Each 9 stands for a digit.) The MAIN-ROUTINE sets K equal to 53 + 4, which is 57. The computer will display:

```
57
```

In the COMPUTE statement, the equal sign and the plus sign must be surrounded by spaces:

Right	Wrong	Wrong
`COMPUTE K = 53 + 4.`	`COMPUTE K=53+4.`	`COMPUTE K = 53+4.`

If you change the picture to 999, K will be a number having three digits. It will be 057 instead of 57. The computer will display:

```
057
```

(Exception: PDP-10 and PDP-20 computers are lazy; they don't bother to display the 0 at the left; they display just 57.)

If you change the picture to 9, K will be a number having only one digit. So K will not be 57. The computer will not display the correct sum.

Like FORTRAN, COBOL uses these operators:

Operator	Meaning
+	plus
-	minus
*	times
/	divided by
**	exponent

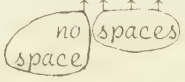
You must put a blank space before and after each operator:

Right	Wrong
`53 + 4`	`53+4`
`7 ** 2`	`7**2`
`- J + 3`	`-J + 3`

Like BASIC, FORTRAN, and PASCAL, COBOL lets you use parentheses. Do *not* put a space after a left parenthesis:

Right

```
COMPUTE K = I * (- J + 3)
```



You can use these short cuts:

Sentence	Short cut
`COMPUTE A = A + 7.`	`ADD 7 TO A.`
`COMPUTE B = B - 4.`	`SUBTRACT 4 FROM B.`

Operators are allowed only in sentences that say COMPUTE, IF, UNTIL, or WHEN:

Allowed	Not allowed	Not allowed	Not allowed
`COMPUTE A = 2 * 3.`	`DISPLAY 2 * 3.`	`MOVE 2 * 3 TO A.`	`ADD 2 * 3 TO A.`

DECIMALS

You can use decimals:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC 9999V99.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        COMPUTE K = 2.208 + 4.109.
        DISPLAY K.
        STOP RUN.
```

The PIC 9999V99 says K is a number having four digits, followed by a decimal point, followed by two digits. (The V stands for the decimal point.) The MAIN-ROUTINE tries to set K equal to 2.208 + 4.109, which is 6.317; but because of K's picture, I will be 0006.31 instead. So the computer should display 0006.31.

PDP-10 and PDP-20 computers don't bother to display the zeros: they display 6.31. PDP-11, IBM, CDC, and Eclipse computers don't bother to display the decimal point: they display 000631.

You can change that program by saying ROUNDED:

```
        COMPUTE K ROUNDED = 2.208 + 4.109.
```

The computer will find the sum (6.317), and round it so K is 0006.32 instead of 0006.31.

If J's picture is 99, saying "COMPUTE J = 200 / 3" makes J be 66. Saying "COMPUTE J ROUNDED = 200 / 3" makes J be 67.

Which is better: saying "COMPUTE X = 4.9" or "MOVE 4.9 TO X"? You should usually say "MOVE 4.9 TO X", because the computer handles it more quickly. But MOVE cannot round; so if you want to round, say COMPUTE.

MOVE can do strange things:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      L PIC 999V99.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE 16725.048 TO L.
        DISPLAY L.
        STOP RUN.
```

L's picture makes the computer move the three digits just left of the decimal point and the two digits just right of it. L will be 725.04.

You can use negative numbers:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC S9999.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        COMPUTE K = 100 - 367.
        DISPLAY K.
        STOP RUN.
```

In K's picture, the S stands for a sign (which can be plus or minus). K will be -0267. The computer should display -0267.

PDP-10 and PDP-20 computers don't bother to display the zero: they display -267. Some other computers display the minus sign on top of the right digit, like this: 026?. On most computers, the minus combines with the 7 and forms a P, like this: 026P.

If you omit the S from K's picture, K will be 0267 instead of -0267.

To find the negative of a power, use parentheses:

```
        COMPUTE A = - (3 ** 2).
```

If you omit the parentheses, the computer will get the wrong answer.

Here's how to translate the BASIC word *INPUT* into FORTRAN, PASCAL, and COBOL:

BASIC	FORTRAN	PASCAL	COBOL
INPUT K	READ *, K	READ(K);	ACCEPT K.

Let's look at a COBOL example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC XXX.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "THIS PROGRAM WANTS YOU TO TYPE SOMETHING".
        ACCEPT K.
        DISPLAY K.
        STOP RUN.
```

The computer displays:

```
THIS PROGRAM WANTS YOU TO TYPE SOMETHING
```

The statement ACCEPT K makes the computer wait for you to type something. If you type—

```
FIGHT
```

the computer will try to move "FIGHT" to K; but since K's picture is XXX, K will be "FIG". The computer will display:

```
FIG
```

If you type—

```
ME
```

the computer will try to move "ME" to K; since K's picture is XXX, K will be "ME ". The computer will display:

```
ME
```

Suppose a program says L PIC 999 and ACCEPT L. If you input—

```
4
```

a PDP-10 or PDP-20 computer will make L be 004, but an IBM or CDC computer will make L be 400.

Suppose a program says M PIC S9999V99 and ACCEPT M. If you want M to be -0034.27, here's what to input:

On PDP-10 and PDP-20 computers	On IBM and CDC computers
-0034.27 *or* -34.27	003427

The computer can edit the output:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC XXXBXXX.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE "HITHER" TO K.
        DISPLAY K.
        STOP RUN.
```

In K's picture, B means a blank space. So when the computer moves "HITHER", K becomes "HIT HER". The computer will display:

```
HIT HER
```

This program displays my phone number:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      PHONE-NUMBER PIC 999B9999.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE 2668128 TO PHONE-NUMBER.
        DISPLAY PHONE-NUMBER.
        STOP RUN.
```

When the computer moves 2668128, PHONE-NUMBER becomes "266 8128". The computer will display:

```
266 8128
```

This program is of historical importance:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      K PIC 9B9B9999.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        COMPUTE K = 741775 + 1.
        DISPLAY "THE DECLARATION OF INDEPENDENCE WAS SIGNED ON " K.
        STOP RUN.
```

The computer will display:

```
THE DECLARATION OF INDEPENDENCE WAS SIGNED ON 7 4 1776
```

Using B to insert a blank is called *editing*. You've learned about four kinds of variables:

Kind of variable	Symbols in picture
string	X
edited string	X B
number	9 V S
edited number	9 B Z , $ * . + - DB CR

You can use these pictures for editing numbers:

K's picture	Meaning	What K will be if you move 50320 to K	What K will be if you move 0 to K
B99999999	a blank followed by eight digits	" 00050320"	" 00000000"
ZZZZZZZZZ	blanks then digits	" 50320"	" 0"
Z,ZZZ,ZZZ	insert commas where appropriate	" 50,320"	" 0"
$,$$$,$$$	put a dollar sign before the digits	" $50,320"	" $0"
*,***,***	put asterisks instead of blanks	"***50,320"	"*********"

To edit decimals, put a decimal point in the picture.

K's picture	If you move 50320.6 to K	If you move .04 to K	If you move 0 to K
Z,ZZZ,ZZZ.ZZ	" 50,320.60"	" .04"	" ."
$,$$$,$$$.$$	" $50,320.60"	" $.04"	" $."
*,***,***.**	"***50,320.60"	"*********.04"	"*********.**"

With those pictures, if you move 0 to K the computer doesn't put any digits in K. To guarantee that K contains a digit, put 9 in the picture:

K's picture	If you move 50320.6 to K	If you move .04 to K	If you move 0 to K
Z,ZZZ,ZZ9.99	" 50,320.60"	" 0.04"	" 0.00"
$,$$$,$$9.99	" $50,320.60"	" $0.04"	" $0.00"
*,***,**9.99	"***50,320.60"	"********0.04"	"********0.00"

To edit negative numbers, use +, -, DB, or CR.

K's picture	Meaning	If you move -2.6 to K	If move 2.6 to K
ZZZ.ZZ+	put - or + afterwards	" 2.60-"	" 2.60+"
ZZZ.ZZ-	put - or blank afterwards	" 2.60-"	" 2.60 "
ZZZ.ZZDB	if negative, put DB (for *debit*)	" 2.60DB"	" 2.60 "
ZZZ.ZZCR	if negative, put CR (for *credit*)	" 2.60CR"	" 2.60 "

For fancier pictures, replace the Z by $, *, or 9.

Here's how to put the sign before the digits:

K's picture	Meaning	If you move -2.6 to K	If move 2.6 to K
+++.++	put - or + before the digits	" -2.60"	" +2.60"
---.--	put - or blank before the digits	" -2.60"	" 2.60"

Here are the differences between numbers and edited numbers:

	Number	Edited number
how to put a decimal point in the picture	V	.
how to put a sign in the picture	S	+, -, DB, or CR
how to fill up most of the picture	9	9, Z, $, *, +, or -
what the value should be	an intermediate result in a long calculation	the final answer to be displayed
If the variable is called K, can you say ACCEPT K?	yes	no
If the variable is called K, can you use K in further computations? For example, can you say COMPUTE L = K + 1?	yes	no
What happens if you try to DISPLAY the value?	might display the zeros, decimal point, and minus sign incorrectly	displays correctly

IF

Like BASIC, FORTRAN, and PASCAL, COBOL uses the words IF and ELSE.

Newest BASIC:

```
INPUT I
IF I>5 THEN J=80: K=90
```

```
INPUT I
IF I>5 THEN J=80: K=90 ELSE J=30: K=50
```

FORTRAN 77:

```
READ *, I
IF (I .GT. 5) THEN
   J=80
   K=90
END IF
```

```
READ *, I
IF (I .GT. 5) THEN
   J=80
   K=90
ELSE
   J=30
   K=50
END IF
```

PASCAL:

```
READ(I);
IF I>5 THEN
        BEGIN;
        J:=80;
        K:=90;
        END;
```

```
READ(I);
IF I>5 THEN
        BEGIN;
        J:=80;
        K:=90;
        END
ELSE
        BEGIN;
        J:=30;
        K:=50;
        END;
```

COBOL:

```
ACCEPT I.
IF I > 5
        MOVE 80 TO J
        MOVE 90 TO K.
```

```
ACCEPT I.
IF I > 5
        MOVE 80 TO J
        MOVE 90 TO K
ELSE
        MOVE 30 TO J
        MOVE 50 TO K.
```

In COBOL, when you write the IF statement, do *not* put a period at the end of every line; instead, put the period just at the end of the entire IF idea.

Notice that I indented the word MOVE. The indentation is optional, but is a good habit. To indent on PDP and Eclipse computers, type a controlled I; to indent on IBM and CDC computers, press the space bar several times.

COBOL uses these IF lines:

IF line	Meaning
IF I = 5	If I is equal to 5
IF I NOT = 5	If I is not equal to 5
IF I > 5	If I is greater than 5
IF I NOT > 5	If I is not greater than 5
IF I < 5	If I is less than 5
IF I NOT < 5	If I is not less than 5

You can use the words AND and OR and abbreviate:

IF line	Abbreviation
IF J > 1 AND J < 100	IF J > 1 AND < 100
IF K < -3 OR K = 6 OR K = 9 OR K = 12 OR K > 50	IF K < -3 OR = 6 OR 9 OR 12 OR > 50

You can say this:

```
IF AGE < 13
        DISPLAY "CHILD"
ELSE IF AGE < 20
        DISPLAY "TEENAGER"
ELSE IF AGE < 40
        DISPLAY "YOUNG ADULT"
ELSE IF AGE < 60
        DISPLAY "MIDDLE-AGED"
ELSE
        DISPLAY "SENIOR CITIZEN".
```

It means, "If AGE is less than 13, display the word CHILD; if not less than 13, do the following: if AGE is less than 20, display the word TEENAGER; if not less than 20, do the following: if AGE is less than 40,..." and so on. The computer will display just one phrase, to describe the person's AGE.

You can say UNTIL:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      I PIC 999.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE 5 TO I.
        PERFORM FIDDLE-WITH-I UNTIL I > 100.
        STOP RUN.
FIDDLE-WITH-I.
        DISPLAY I.
        COMPUTE I = I * 2.
```

The computer will perform FIDDLE-WITH-I repeatedly, until I > 100. The computer will display 5, 10, 20, 40, and 80. It will not display 160.

Here are the details. When the computer encounters PERFORM FIDDLE-WITH-I UNTIL I > 100, it checks whether I > 100. If I > 100, the computer proceeds to the next statement (the STOP RUN); but if I is *not* greater than 100, the computer performs FIDDLE-WITH-I and then re-executes the statement PERFORM FIDDLE-WITH-I UNTIL I > 100.

To translate a BASIC "FOR...NEXT loop" into COBOL, say "PERFORM":

BASIC

```
10 FOR I = 5 TO 17
20     PRINT I
30 NEXT I
```

COBOL

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        PERFORM DISPLAY-IT
                VARYING I FROM 5 BY 1 UNTIL I > 17.
        STOP RUN.
DISPLAY-IT.
        DISPLAY I.
```

BASIC

```
10 FOR I = 5 TO 17 STEP 3
20     PRINT I
30 NEXT I
```

COBOL

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        PERFORM DISPLAY-IT
                VARYING I FROM 5 BY 3 UNTIL I > 17.
        STOP RUN.
DISPLAY-IT.
        DISPLAY I.
```

BASIC

```
10 FOR I = 5 TO 17
20     FOR J = 1 TO 3
30         PRINT I,J
40     NEXT J
50 NEXT I
```

COBOL

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        PERFORM DISPLAY-IT
                VARYING I FROM 5 BY 1 UNTIL I > 17
                AFTER VARYING J FROM 1 BY 1 UNTIL J > 3.
        STOP RUN.
DISPLAY-IT.
        DISPLAY I J.
```

GO TO

Like BASIC, FORTRAN, and PASCAL, COBOL lets you say GO TO. In COBOL, put a space between GO and TO. (In PASCAL, you do *not* put a space between GO and TO.)

For example, instead of saying STOP RUN, you can say GO TO MAIN-ROUTINE:

```
PROCEDURE DIVISION.
MAIN-ROUTINE.
        DISPLAY "WHEATIES".
        DISPLAY "ARE WONDERFUL".
        GO TO MAIN-ROUTINE.
```

The computer will display "WHEATIES" and "ARE WONDERFUL", repeatedly:

```
WHEATIES
ARE WONDERFUL
WHEATIES
ARE WONDERFUL
WHEATIES
ARE WONDERFUL
etc.
```

The main routine can consist of *two* paragraphs, called MAIN-ROUTINE-BEGINNING and MAIN-ROUTINE-LOOP:

```
PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        DISPLAY "PLEASE".
MAIN-ROUTINE-LOOP.
        DISPLAY "KISS".
        DISPLAY "ME".
        GO TO MAIN-ROUTINE-LOOP.
```

The computer will display PLEASE, then repeatedly display KISS and ME:

```
PLEASE
KISS
ME
KISS
ME
KISS
ME
etc.
```

The main routine can consist of *three* paragraphs, called MAIN-ROUTINE-BEGINNING, MAIN-ROUTINE-LOOP, and MAIN-ROUTINE-ENDING:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      HUMAN-RESPONSE PIC XXX.

PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        DISPLAY "I WILL RECITE A SHORT POEM".
MAIN-ROUTINE-LOOP.
        DISPLAY " ".
        DISPLAY "YOUR NOSE".
        DISPLAY "BLOWS".
        DISPLAY " ".
        DISPLAY "WOULD YOU LIKE TO HEAR THE POEM AGAIN?".
        ACCEPT HUMAN-RESPONSE.
        IF HUMAN-RESPONSE = "YES"
                GO TO MAIN-ROUTINE-LOOP.
MAIN-ROUTINE-ENDING.
        DISPLAY "YOU HAVE BEEN A GREAT AUDIENCE".
        STOP RUN.
```

When you run that program, the computer says:

```
I WILL RECITE A SHORT POEM
```

Then it recites the program:

```
YOUR NOSE
BLOWS
```

Then it asks:

```
WOULD YOU LIKE TO HEAR THE POEM AGAIN?
```

If you say YES, the computer repeats the poem, and then asks you whether you'd like to hear it a third time. If you say YES again, the computer recites the poem a third time, and then asks whether you'd like to hear it a *fourth* time. The computer recites the poem repeatedly, until you finally stop saying YES. Then the computer says—

```
YOU HAVE BEEN A GREAT AUDIENCE
```

and stops.

In that program, if you don't say YES, the computer doesn't repeat the poem. So if you don't say YES, the computer acts as if you said NO. The following version is an improvement; if you don't say YES, and you don't say NO, the computer asks the question again:

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01      HUMAN-RESPONSE PIC XXX.

  PROCEDURE DIVISION.
  MAIN-ROUTINE-BEGINNING.
          DISPLAY "I WILL RECITE A SHORT POEM".
  MAIN-ROUTINE-LOOP.
          DISPLAY " ".
          DISPLAY "YOUR NOSE".
          DISPLAY "BLOWS".
          DISPLAY " ".
→→        PERFORM GET-HUMAN-RESPONSE.
          IF HUMAN-RESPONSE = "YES"
                  GO TO MAIN-ROUTINE-LOOP.
  MAIN-ROUTINE-ENDING.
          DISPLAY "YOU HAVE BEEN A GREAT AUDIENCE".
          STOP RUN.
→→GET-HUMAN-RESPONSE.
→→        DISPLAY "WOULD YOU LIKE TO HEAR THE POEM AGAIN?".
→→        ACCEPT HUMAN-RESPONSE.
→→        IF HUMAN-RESPONSE NOT = "YES" AND NOT = "NO"
→→                DISPLAY "PLEASE SAY YES OR NO!"
→→                GO TO GET-HUMAN-RESPONSE.
```

GO TO resembles PERFORM. Here's the difference between GO TO and PERFORM....

To go to a *different routine*, say *PERFORM*. To go to a different paragraph in the *same* routine, say *GO TO*.

For example, suppose you want to go from MAIN-ROUTINE-BEGINNING to FUNNY-SUBROUTINE; since you're going to a *different routine*, say *PERFORM*.

Suppose you want to go from MAIN-ROUTINE-BEGINNING to MAIN-ROUTINE-ENDING; since you're going to a different paragraph in the *same* routine, say *GO TO*.

THRU

Like the main routine, a subroutine can consist of several paragraphs. For example, subroutine FUNNY-FACE can consist of three paragraphs, called FUNNY-FACE-BEGINNING, FUNNY-FACE-LOOP, and FUNNY-FACE-ENDING. To make the computer do the entire subroutine, say:

```
PERFORM FUNNY-FACE-BEGINNING THRU FUNNY-FACE-ENDING.
```

STRUCTURES

FILES

To manipulate a file whose name is POEM, fill in the blanks:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        The program's name.
AUTHOR.
        Your name.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT POEM-FILE ASSIGN TO the file's location.

DATA DIVISION.
FILE SECTION.
FD      POEM-FILE how the file is labeled.
01      POEM-LINE PIC a picture of a line of the file.
WORKING-STORAGE SECTION.
A description of each variable that's not in the file.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        OPEN output or input POEM-FILE.
        What you want to do to the file.
        CLOSE POEM-FILE.
        STOP RUN.
```

Like every COBOL program, that outline consists of four divisions: the IDENTIFICATION DIVISION, the ENVIRONMENT DIVISION, the DATA DIVISION, and the PROCEDURE DIVISION. Let's look at each division.

Identification division

The IDENTIFICATION DIVISION consists of two paragraphs: the PROGRAM-ID and the AUTHOR. For the PROGRAM-ID, fill in the program's name, *which must be different than the name of the file*. Since the name of the file is POEM, the name of the program must *not* be POEM. If you're lazy, you can omit the AUTHOR.

Environment division

The ENVIRONMENT DIVISION consists of two sections: the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION.

The CONFIGURATION SECTION consists of two paragraphs: the SOURCE-COMPUTER and the OBJECT-COMPUTER. On CDC and PDP-11 computers, the CONFIGURATION SECTION is required; but on IBM, Eclipse, PDP-10, and PDP-20 computers, the entire CONFIGURATION SECTION is optional, so you can abridge the ENVIRONMENT DIVISION:

```
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT POEM-FILE ASSIGN TO the file's location.
```

The INPUT-OUTPUT SECTION consists of just one paragraph, which is the FILE-CONTROL. The FILE-CONTROL paragraph consists of a sentence that says SELECT, then the file's name (POEM FILE), then ASSIGN TO, and finally a blank (which you must fill in, and which tells the file's location).

What do you put in that blank? The answer depends on the file's location. Is the file on a disk? On punched cards? Or on paper produced by the printer? Here's what to put in the blank, for various computers:

	Printer	Card reader	Disk
Eclipse	PRINTER	"$CDR"	"POEM"
CDC	OUTPUT	INPUT	POEM
PDP-11	"LP:"	"CR:"	"DK:"
PDP-10, PDP-20	LPT	CDR	DSK RECORDING MODE ASCII
IBM using OS	UT-S-POEM	UT-S-POEM	UT-S-POEM
IBM using DOS	SYS006-UR-1403-S	SYS005-UR-2540R-S	SYS020-UT-3330-S-POEM

Data division

The DATA DIVISION consists of two sections: the FILE SECTION and the WORKING-STORAGE SECTION.

At the beginning of the FILE SECTION, say FD (which means "File Description"). To the right of the FD, say POEM-FILE, and then fill in the blank, which tells how the file is labeled:

Computer	What to put in the blank
CDC	LABEL RECORDS ARE OMITTED
PDP-11: disk	LABEL RECORDS ARE STANDARD VALUE OF ID "POEM"
PDP-11: printer or card reader	LABEL RECORDS ARE OMITTED
PDP-10, PDP-20	VALUE ID "POEM␣␣␣␣␣"
IBM OS	LABEL RECORDS ARE STANDARD
IBM DOS: disk	LABEL RECORDS ARE STANDARD
IBM DOS: printer or card reader	LABEL RECORDS ARE OMITTED

On PDP-10 and PDP-20 computers, put enough blank spaces (␣) after POEM so that the string has 9 characters. On Eclipse computers, do *not* fill in the blank; just say:

```
FD      POEM-FILE.
```

Underneath the line that says FD, you must say 01. The 01 line includes a picture of a line of the file. For example, if a line of the file is an 80-character string, the 01 line should say:

```
01      POEM-LINE PIC X(80).
```

The WORKING-STORAGE SECTION describes each variable that's not in the file.

Procedure division

The PROCEDURE DIVISION's MAIN-ROUTINE should begin with the word OPEN, and end with the words CLOSE and STOP RUN. In the OPEN statement, you can say either—

```
        OPEN OUTPUT POEM-FILE.
```

or:

```
        OPEN INPUT POEM-FILE.
```

If you say OPEN OUTPUT POEM-FILE, the computer will output to the POEM-FILE; so it will copy information from the RAM to the POEM-FILE. If you say OPEN INPUT POEM-FILE, the computer will input from the POEM-FILE; so it will copy information from the POEM-FILE to the RAM.

In the PROCEDURE DIVISION, when you fill in the blank about "what you want to do to the file", you must say either WRITE POEM-LINE or READ POEM-FILE. If the file is OPEN OUTPUT (which means you're copying from the RAM to the file), say WRITE POEM-LINE; if the file is OPEN INPUT (which means you're copying from the file to the RAM), say READ POEM-FILE.

Here's a poetic masterpiece:

```
CANDY IS DANDY
BUT LIKKER IS QUIKKER
```

It was composed by the famous poet Ogden Nash. Let's make the computer write that masterpiece onto a disk, and make the masterpiece become a file named POEM.

Here's how:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        CANDY.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT POEM-FILE ASSIGN TO the file's location.

DATA DIVISION.
FILE SECTION.
FD      POEM-FILE how the file is labeled.
01      POEM-LINE PIC X(21).     Therefore, each line will have 21 characters, like this
                                 CANDY IS DANDY■■■■■■■
PROCEDURE DIVISION.              BUT LIKKER IS QUIKKER
MAIN-ROUTINE.
        OPEN OUTPUT POEM-FILE.                  The computer creates POEM-FILE for output.
        MOVE "CANDY IS DANDY" TO POEM-LINE.     POEM-LINE is "CANDY IS DANDY■■■■■■■".
        WRITE POEM-LINE.                        That POEM-LINE is copied onto the file.
        MOVE "BUT LIKKER IS QUIKKER" TO POEM-LINE.  POEM-LINE becomes a new string.
        WRITE POEM-LINE.                        That new POEM-LINE is copied onto the file
        CLOSE POEM-FILE.
        STOP RUN.
```

That program doesn't require a WORKING-STORAGE SECTION, so I omitted it. Since I was lazy, I also omitted the AUTHOR paragraph. When you run that program, the computer will create a file on disk; the file will be called POEM; it will contain this message:

```
CANDY IS DANDY■■■■■■■
BUT LIKKER IS QUIKKER
```

Reading

This program reads the file POEM, and displays it on your terminal:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        READER.

The ENVIRONMENT DIVISION and DATA DIVISION are the same as the previous program's.

PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        OPEN INPUT POEM-FILE.       Find POEM on the disk, & prepare to input from it.
MAIN-ROUTINE-LOOP.
        READ POEM-FILE AT END GO TO MAIN-ROUTINE-ENDING.  Read a line from POEM-FILE.
        DISPLAY POEM-LINE.          Display that line, so you see it on your terminal.
        GO TO MAIN-ROUTINE-LOOP.    Go back to read another line.
MAIN-ROUTINE-ENDING.
        DISPLAY "THAT WAS THE WHOLE POEM".
        CLOSE POEM-FILE.
        STOP RUN.
```

In the MAIN-ROUTINE-LOOP, the first line means: try to READ a line from POEM-FILE; but if a line can*not* be read (because the file has ended), go to MAIN-ROUTINE-ENDING instead.

The READ statement differs from the WRITE statement in two ways:

A WRITE statement mentions a LINE, but a READ statement mentions a FILE.
A READ statement must contain the words AT END.

Counting

This program reads a file called POEM, counts how many lines are in it, and displays the count:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        COUNTS.

The ENVIRONMENT DIVISION is the same as the previous program's.

DATA DIVISION.
FILE SECTION.
FD      POEM-FILE how the file is labeled.
01      POEM-LINE PIC X(21).                  assumes POEM-LINE has 21 characters
WORKING-STORAGE SECTION.
01      COUNT-OF-HOW-MANY-LINES PIC 99.       assumes the count is a two-digit number;
                                              so assumes POEM has less than 100 lines

PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        OPEN INPUT POEM-FILE.                 Find POEM on the disk.
        MOVE 0 TO COUNT-OF-HOW-MANY-LINES.    Start the count at 0.
MAIN-ROUTINE-LOOP.
        READ POEM-FILE AT END GO TO MAIN-ROUTINE-ENDING.  Read a line from POEM-FILE.
        ADD 1 TO COUNT-OF-HOW-MANY-LINES.     Add 1 to the count.
        GO TO MAIN-ROUTINE-LOOP.              Go to read another line.
MAIN-ROUTINE-ENDING.                          When the file is done,
        DISPLAY COUNT-OF-HOW-MANY-LINES.      display the count.
        CLOSE POEM-FILE.
        STOP RUN.
```

Copying

This program reads a file called POEM, and creates a copy of it; the copy is a file called POEM2:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        COPIER.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT POEM-FILE ASSIGN TO the location of POEM.
        SELECT POEM2-FILE ASSIGN TO the location of POEM-2.

DATA DIVISION.
FILE SECTION.
FD      POEM-FILE the labeling for POEM.
01      POEM-LINE PIC X(21).
FD      POEM2-FILE the labeling for POEM2.
01      POEM2-LINE PIC X(21).

PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        OPEN INPUT POEM-FILE.           The computer will input from POEM-FILE,
        OPEN OUTPUT POEM2-FILE.         and output to POEM2-FILE.
MAIN-ROUTINE-LOOP.                      The computer does the following repeatedly:
        READ POEM-FILE AT END GO TO MAIN-ROUTINE-ENDING.  read a line from POEM-FILE;
        MOVE POEM-LINE TO POEM2-LINE.   copy that line to POEM2-LINE;
        WRITE POEM2-LINE.               and write POEM2-LINE onto POEM2-FILE.
        GO TO MAIN-ROUTINE-LOOP.
MAIN-ROUTINE-ENDING.
        DISPLAY "THE FILE HAS BEEN COPIED".
        CLOSE POEM-FILE POEM2-FILE.
        STOP RUN.
```

Pictures

Suppose you're dealing with a file named JOE, and each line of JOE-FILE is a three-digit number. Should the line's picture be edited (JOE-LINE PIC ZZZ) or unedited (JOE-LINE PIC 999)?

When you read a file, the line's picture must be unedited and match the picture in the program that wrote the file.

When you write a file, ask yourself, "What will read it?" If the answer is "a COBOL program", the picture must be unedited. If the answer is "only a human", edit the picture.

Remember: if one program writes a file, and another program reads it, both programs must use the same picture. For example, if a program writes JACK-FILE and says JACK-LINE PIC S9999V99, the program that reads JACK-FILE must also say JACK-LINE PIC S9999V99.

Peculiarities

To write and read unedited numbers, the computer takes a short-cut: it omits decimal points, and locates the negative sign on top of the last digit. For example, instead of writing -0034.27 in JACK-FILE, the computer writes just 003427. When another program reads 003427 from the file, the S9999V99 picture tells the computer the 003427 means -0034.27.

After you WRITE a line, you cannot use the line again in the program. For example, after you say WRITE POEM-LINE, you should not say MOVE POEM-LINE TO K; it won't work.

Multiple widths

Let's make the computer compute the square of 12 and the square of 13 and write this file:

```
HERE ARE THE SQUARES:
144
169
THEY WERE REAL GROOVY
```

The top and bottom lines are long strings whose pictures are X(21). The other two lines are short numbers whose pictures are 999. Here's the program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        SQUARE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT REPORT-FILE ASSIGN TO the location of REPORT.  The file is named REPORT.

DATA DIVISION.
FILE SECTION.
FD      REPORT-FILE the labeling for REPORT.
01      REPORT-LINE PIC X(21).          REPORT-LINE is a 21-character string.
01      REPORT-LINE2 PIC 999.           REPORT-LINE2 is a 3-digit number.

PROCEDURE DIVISION.
MAIN-ROUTINE.
        OPEN OUTPUT REPORT-FILE.        The computer will create a file named REPORT.
        MOVE "HERE ARE THE SQUARES:" TO REPORT-LINE.
        WRITE REPORT-LINE.              The computer writes "HERE ARE THE SQUARES:".
        COMPUTE REPORT-LINE2 = 12 * 12. REPORT-LINE2 is 144.
        WRITE REPORT-LINE2.             The computer writes 144.
        COMPUTE REPORT-LINE2 = 13 * 13. REPORT-LINE2 is 169.
        WRITE REPORT-LINE2.             The computer writes 169.
        MOVE "THEY WERE REAL GROOVY" TO REPORT-LINE.
        WRITE REPORT-LINE.              The computer writes "THEY WERE REAL GROOVY".
        CLOSE REPORT-FILE.
        STOP RUN.
```

GROUP ITEMS

In the data division, you can say:

```
01      K.
        02      L PIC 999.
        02      M PIC 9.
        02      N PIC 99.
```

That means K is a combination of L, M, and N. If the procedure division says—

```
MOVE 427 TO L.
MOVE 8 TO M.
MOVE 31 TO N.
```

then K will be "427831".

Since K is a combination of other variables, K is called a *group variable* or *group item*. L, M, and N are *elementary items*. Notice that K is the string "427831", not the number 427831. A group item is always a string. Since K is a string, not a number, you cannot say ADD 1 TO K, although you can say ADD 1 TO L or ADD 1 TO M or ADD 1 TO N.

Here's a group item, for a weight-reducing studio:

```
01      PERSONAL-INFO-ABOUT-CLIENT.
        02      CLIENT-NAME.
                03      FIRST-NAME PIC X(15).
                03      MIDDLE-INITIAL PIC X.
                03      LAST-NAME PIC X(20).
        02      CLIENT-SEX PIC X.              A person's sex is "M" or "F".
        02      CLIENT-AGE PIC 99.
        02      WEIGHT-PROGRESS.
                03      WEIGHT-WHEN-ENTERED-PROGRAM PIC 999.
                03      WEIGHT-THIS-WEEK PIC 999.
                03      NUMBER-OF-WEEKS-SO-FAR PIC 999.
```

PERSONAL-INFO-ABOUT-CLIENT is composed of CLIENT-NAME (which is composed of FIRST-NAME, MIDDLE-INITIAL, and LAST-NAME), CLIENT-SEX, and WEIGHT-PROGRESS (which is composed of WEIGHT-WHEN-ENTERED-PROGRAM, WEIGHT-THIS-WEEK, and NUMBER-OF-WEEKS-SO-FAR). So PERSONAL-INFO-ABOUT-CLIENT is composed of numbers and strings. Altogether, PERSONAL-INFO-ABOUT-CLIENT contains 48 characters (15 + 1 + 20 + 1 + 2 + 3 + 3 + 3); the computer considers PERSONAL-INFO-ABOUT-CLIENT to be a string whose picture is X(48). If you say L PIC X(48), you can move all the PERSONAL-INFO-ABOUT-CLIENT to L by saying:

```
MOVE PERSONAL-INFO-ABOUT-CLIENT TO L.
```

To move the CLIENT-NAME to M, without moving the CLIENT-SEX, CLIENT-AGE, and WEIGHT-PROGRESS, say:

```
MOVE CLIENT-NAME TO M.
```

To write lots of information into a file, make the file's LINE be a group item.

How to extract from a file

Suppose you've already created a file whose name is EMPLOY; it's on disk or cards. Suppose the file contains information about employees. Suppose each line of the file contains 80 characters, as follows....

Characters 1-40: the employee's name
Characters 61-70: the employee's home phone number, including the area code
The other characters (41-60 and 71-80): "miscellaneous information", such as the employee's age, sex, address, salary, kind of job, and number of years with the company

Let's create a new file, called REPORT, on disk or on the printer's paper. Let's make REPORT contain just the employees' names and phone numbers, and omit the "miscellaneous information". Here's how:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        PHONES.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT EMPLOY-FILE ASSIGN TO the location of EMPLOY.
        SELECT REPORT-FILE ASSIGN TO the location of REPORT.

DATA DIVISION.
FILE SECTION.
FD      EMPLOY-FILE the labeling for EMPLOY.
01      EMPLOY-LINE.
        02      EMPLOYEE-NAME PIC X(40).
        02      FILLER PIC X(20).
        02      HOME-PHONE.
                03      AREA-CODE PIC 999.
                03      PHONE-EXCHANGE PIC 999.
                03      REST-OF-PHONE-NUMBER PIC 9999.
        02      FILLER PIC X(10).
FD      REPORT-FILE the labeling for REPORT.
01      REPORT-LINE.
        02      EMPLOYEE-NAME-REPORTED PIC X(40).
        02      HOME-PHONE-REPORTED.
                03      LEFT-PARENTHESIS PIC X.
                03      AREA-CODE-REPORTED PIC 999.
                03      RIGHT-PARENTHESIS PIC X.
                03      PHONE-EXCHANGE-REPORTED PIC B999.
                03      THE-DASH PIC X.
                03      REST-OF-PHONE-NUMBER-REPORTED PIC 9999.

PROCEDURE DIVISION.
MAIN-ROUTINE-BEGINNING.
        OPEN INPUT EMPLOY-FILE.
        OPEN OUTPUT REPORT-FILE.
MAIN-ROUTINE-LOOP.
        READ EMPLOY-FILE AT END GO TO MAIN-ROUTINE-ENDING.
        MOVE EMPLOYEE-NAME TO EMPLOYEE-NAME-REPORTED.
        MOVE "(" TO LEFT-PARENTHESIS.
        MOVE AREA-CODE TO AREA-CODE-REPORTED.
        MOVE ")" TO RIGHT-PARENTHESIS.
        MOVE PHONE-EXCHANGE TO PHONE-EXCHANGE-REPORTED.
        MOVE "-" TO THE-DASH.
        MOVE REST-OF-PHONE-NUMBER TO REST-OF-PHONE-NUMBER-REPORTED.
        WRITE REPORT-LINE.
        GO TO MAIN-ROUTINE-LOOP.
MAIN-ROUTINE-ENDING.
        CLOSE EMPLOY-FILE REPORT-FILE.
        STOP RUN.
```

Throughout the DATA DIVISION, the special word "FILLER" stands for data the program won't mention later.

Characters 1-40 are EMPLOYEE-NAME.
Characters 41-60 are irrelevant.
Characters 61-63 are AREA-CODE.
Characters 64-66.
Characters 67-70.
Characters 71-80 are irrelevant.

The PICs say HOME-PHONE looks like this— 6172668128 but make HOME-PHONE-REPORTED look like this: (617) 266-8128

The MAIN-ROUTINE-LOOP reads a line from EMPLOY-FILE, copies data into each part of REPORT-LINE, and then writes REPORT-LINE.

Suppose CUSTOM is a disk file that contains information about your customers. Suppose each line of the file contains 80 characters, as follows....

Characters 1-20: the customer's last name
Characters 21-80: other information about the customer

Here's how to put the file in alphabetical order, according to the customer's name:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        ALPHA.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT CUSTOM-FILE ASSIGN TO the location of CUSTOM.
        SELECT SORT-FILE ASSIGN TO the location of SORT.

DATA DIVISION.
FILE SECTION.
FD      CUSTOM-FILE the labeling for CUSTOM.
01      CUSTOM-LINE PIC X(80).
SD      SORT-FILE.
01      SORT-LINE.
        02      LAST-NAME PIC X(20).
        02      FILLER PIC X(60).

PROCEDURE DIVISION.
MAIN-ROUTINE.
        SORT SORT-FILE
                ASCENDING KEY LAST-NAME
                USING CUSTOM-FILE
                GIVING CUSTOM-FILE.
        STOP RUN.
```

Putting a file in order, by alphabetizing or any other method, is called *sorting*. To sort the CUSTOM-FILE, the computer has to create a temporary disk file called a SORT-FILE.

In the sentence that says SELECT SORT-FILE, here's what to put for "the location of SORT":

Computer	The location of SORT
Eclipse	"SORT"
CDC	SORT
PDP-10, PDP-20	DSK DSK DSK RECORDING MODE ASCII
IBM using OS	UT-S-POEM
IBM using DOS	SYS001-UT-3330-S-SORTWK1

In the DATA DIVISION's FILE SECTION, the SD means a Sort-file Description. In the PROCEDURE DIVISION, the SORT sentence makes the computer automatically open the CUSTOM-FILE, sort it, and close it.

In the SORT sentence, if you replace ASCENDING by DESCENDING, the computer will sort the file in reverse order, so the Z's come first and the A's come last.

You can make the program fancier, by inserting extra statements before and after the SORT statement. But since the SORT statement automatically tells the computer to open CUSTOM-FILE, the CUSTOM-FILE must not be open already. If you already said OPEN CUSTOM-FILE, you must say CLOSE CUSTOM-FILE before you give the SORT statement.

If you replace GIVING CUSTOM-FILE by GIVING REPORT-FILE, the computer won't change CUSTOM-FILE, but will create a REPORT-FILE containing the information sorted. For REPORT-FILE, you must type an FD and SELECT it. The computer will automatically open it, so it must not be open already.

Within each line of CUSTOM-FILE, suppose characters 51-57 tell how much the customer bought from you during the past year. Let's find out which customers bought the most. Let's make the computer print the customer that bought the most, then the customer that bought the next most, etc. If two customers bought exactly the same amount, let's make the computer print their names in alphabetical order. This program does it:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        BIGBUY.

The ENVIRONMENT DIVISION is the same as the previous program's.

DATA DIVISION.
FILE SECTION.
FD      CUSTOM-FILE the labeling for CUSTOM.
01      CUSTOM-LINE PIC X(80).
SD      SORT-FILE.
01      SORT-LINE.
        02      LAST-NAME PIC X(20).                    characters 1-20
        02      FILLER PIC X(30).                       characters 21-50
        02      AMOUNT-BOUGHT-DURING-YEAR PIC 99999V99. characters 51-57 (the V doesn't
        02      FILLER PIC X(23).                                          use a column)

PROCEDURE DIVISION.
MAIN-ROUTINE.
        SORT SORT-FILE
                DESCENDING KEY AMOUNT-BOUGHT-DURING-YEAR
                ASCENDING KEY LAST-NAME
                USING CUSTOM-FILE
                GIVING CUSTOM-FILE.
        STOP RUN.
```

The SORT sentence says: sort the file so that AMOUNT-BOUGHT-DURING-YEAR is in DESCENDING order; in case of a tie, put LAST-NAME is ASCENDING order.

Suppose OLDCUS and NEWCUS are files (on disk or cards), as follows....

OLDCUS describes your old customers, and NEWCUS describes your newer customers. In those files, each line contains 80 characters; characters 1-20 contain the customer's last name. Each file is already in alphabetical order, by customer's last name.

Let's combine the two files. In other words, let's create a "combination" file (on disk or printer paper), called ALLCUS, that contains *all* the customers; and let's make ALLCUS be in alphabetical order also. Here's how:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.
        MERGER.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OLDCUS-FILE ASSIGN TO the location of OLDCUS.
        SELECT NEWCUS-FILE ASSIGN TO the location of NEWCUS.
        SELECT ALLCUS-FILE ASSIGN TO the location of ALLCUS.
        SELECT SORT-FILE ASSIGN TO the location of SORT.

DATA DIVISION.
FILE SECTION.
FD      OLDCUS-FILE the labeling for OLDCUS.
01      OLDCUS-LINE PIC X(80).
FD      NEWCUS-FILE the labeling for NEWCUS.
01      NEWCUS-LINE PIC X(80).
FD      ALLCUS-FILE the labeling for ALLCUS.
01      ALLCUS-LINE PIC X(80).
SD      SORT-FILE.
01      SORT-LINE.
        02      LAST-NAME PIC X(20).
        02      FILLER PIC X(60).

PROCEDURE DIVISION.
MAIN-ROUTINE.
        MERGE SORT-FILE
                ASCENDING KEY LAST-NAME
                USING OLDCUS-FILE NEWCUS-FILE
                GIVING ALLCUS-FILE.
        STOP RUN.
```

That program creates ALLCUS, which is a combination of OLDCUS and NEWCUS. To do that, the computer must create a SORT-FILE.

The word MERGE automatically opens and closes all the files involved. So do *not* say OPEN or CLOSE.

Warning: the word MERGE is in COBOL 74 but not in COBOL 68. So if your computer is old fashioned and understands just COBOL 68, it doesn't understand the word MERGE.

If your computer understands the word MERGE, you can merge as many files as you like. For example, if you have files called CUS1, CUS2, CUS3, and CUS4, you can say:

```
        MERGE SORT-FILE
                ASCENDING KEY LAST-NAME
                USING CUS1-FILE CUS2-FILE CUS3-FILE CUS4-FILE
                GIVING ALLCUS-FILE.
```

Before you MERGE, make sure that the files you're USING are already in alphabetical order.

SUBSCRIPTS

Like BASIC, FORTRAN, and PASCAL, COBOL lets you use subscripts.

For example, suppose your 4 favorite friends are SUE, JOE, TOM, and ANN. Let's make FAVORITE-FRIEND (1) be "SUE", FAVORITE-FRIEND (2) be "JOE", FAVORITE-FRIEND (3) be "TOM", and FAVORITE-FRIEND (4) be "ANN". Here's how:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01      FAVORITE-FRIEND-TABLE.
        02      FAVORITE-FRIEND OCCURS 4 TIMES PIC XXX.  You have 4 FAVORITE-FRIENDs;
                                                         each has PIC XXX.
PROCEDURE DIVISION.
MAIN-ROUTINE.
        MOVE "SUE" TO FAVORITE-FRIEND (1).
        MOVE "JOE" TO FAVORITE-FRIEND (2).
        MOVE "TOM" TO FAVORITE-FRIEND (3).
        MOVE "ANN" TO FAVORITE-FRIEND (4).
        DISPLAY FAVORITE-FRIEND (1).
        DISPLAY FAVORITE-FRIEND (2).
        DISPLAY FAVORITE-FRIEND (3).
        DISPLAY FAVORITE-FRIEND (4).
        STOP RUN.
```

The computer will display:

```
SUE
JOE
TOM
ANN
```

When typing the program, remember to put a blank space before the subscript:

```
FAVORITE-FRIEND (1)
               ↑
          blank space
```

In COBOL, you say "OCCURS" instead of "DIMENSION":

BASIC	FORTRAN	PASCAL	COBOL
DIM F(4)	DIMENSION F(4)	F: ARRAY [1..4]	F OCCURS 4 TIMES

The subscript can be a variable. For example, instead of saying—

```
DISPLAY FAVORITE-FRIEND (1).
DISPLAY FAVORITE-FRIEND (2).
DISPLAY FAVORITE-FRIEND (3).
DISPLAY FAVORITE-FRIEND (4).
```

you can say:

```
DISPLAY FAVORITE-FRIEND (I).
```

To do that, you must tell the computer that the I goes from 1 to 4. Here's how:

```
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01      FAVORITE-FRIEND-TABLE.
          02      FAVORITE-FRIEND OCCURS 4 TIMES PIC XXX.
→→01      I PIC 9.                                    I is a one-digit number.

  PROCEDURE DIVISION.
  MAIN-ROUTINE.
          MOVE "SUE" TO FAVORITE-FRIEND (1).
          MOVE "JOE" TO FAVORITE-FRIEND (2).
          MOVE "TOM" TO FAVORITE-FRIEND (3).
          MOVE "ANN" TO FAVORITE-FRIEND (4).
→→        PERFORM SHOW-FRIENDSHIP
→→                VARYING I FROM 1 BY 1 UNTIL I > 4.  I will be 1, 2, 3, 4.
          STOP RUN.
→→SHOW-FRIENDSHIP.
→→        DISPLAY FAVORITE-FRIEND (I).                I is the subscript.
```

To make the program run faster, say "COMP" at the end of the subscript's picture:

Computer	What to say
PDP, Eclipse	01 I PIC 9 COMP.
CDC	01 I PIC 9 COMP-1.
IBM	01 I PIC 9 COMP SYNC.

COMP stands for the word COMPUTATIONAL; SYNC stands for the word SYNCHRONIZED.

A subscript cannot contain an operation:

Okay	Wrong
FAVORITE-FRIEND (3)	FAVORITE-FRIEND (2 + 1)
	↑
	The + is not allowed.

Here's how to make Y-TABLE be a table that has 4 rows and 6 columns:

```
01      Y-TABLE.
        02      Y-ROW OCCURS 4 TIMES.
                03      Y OCCURS 6 TIMES PIC XXX.
```

The entire table is called:

Y-TABLE

The first row of Y-TABLE is called:

Y-ROW (1)

The second row of Y-TABLE is called:

Y-ROW (2)

The entry in the 2nd row and 5th column of Y-TABLE is called:

Y (2, 5)

spaces

Suppose you teach 25 students, you've given each student 4 tests, and you want to store everything in a table. You want the table to contain 25 rows (a row for each student); in each row, you want the student's first name, middle initial, last name, and 4 scores. Here's how:

```
01      STUDENT-INFORMATION-TABLE.
        02      STUDENT-INFORMATION-ROW OCCURS 25 TIMES.
                03      FIRST-NAME PIC X(15).
                03      MIDDLE-INITIAL PIC X.
                03      LAST-NAME PIC X(20).
                03      TEST-SCORE OCCURS 4 TIMES PIC 999.
```

The entire table is called:

```
STUDENT-INFORMATION-TABLE
```

The table contains 25 rows; the first row is called:

```
STUDENT-INFORMATION-ROW (1)
```

The twelfth row is called:

```
STUDENT-INFORMATION-ROW (12)
```

The information in the twelfth row is called:

```
FIRST-NAME (12)
MIDDLE-INITIAL (12)
LAST-NAME (12)
TEST-SCORE (12, 1)
TEST-SCORE (12, 2)
TEST-SCORE (12, 3)
TEST-SCORE (12, 4)
```

EXOTIC FEATURES

IDENTIFICATION DIVISION

The IDENTIFICATION DIVISION can include these paragraphs:

```
PROGRAM-ID.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
SECURITY.
```

In each paragraph after the PROGRAM-ID, put whatever garbage you please. The computer ignores everything the IDENTIFICATION DIVISION says.

The IDENTIFICATION DIVISION helps the computer center's librarian classify your program. The librarian wants the INSTALLATION paragraph to contain the computer center's name and address, the DATE-WRITTEN paragraph to tell when you finished debugging the program, the DATE-COMPILED paragraph to tell when the computer translated the program from COBOL into machine language, and the SECURITY paragraph to tell who may look at the program and who must not.

If you put the wrong date in the DATE-COMPILED paragraph, don't worry: when you ask the computer to produce a *COBOL listing* of your program, the listing will automatically show the correct date instead.

COMMENTS

The computer ignores any line that begins with an asterisk. So if you put this line in your program—

```
*THIS IS A LOUSY PROGRAM
```

— the computer will ignore the comment.

Create comments that explain how your program works. Put the comments near the bottom of the IDENTIFICATION DIVISION, near the top of the PROCEDURE DIVISION, and wherever your program looks confusing.

On PDP and Eclipse computers, put the asterisk at the far left; don't put any blank spaces before the asterisk. On IBM and CDC computers, put six blank spaces before the asterisk, so that the asterisk is in column 7.

DECIMAL-POINT IS COMMA

Some Europeans write commas instead of decimal points, and write decimal points instead of commas.

United States and England: 5,243,794.95
France and Italy: 5.243.794,95
Germany: 5 243 794,95

To write a COBOL program for a Frenchman, an Italian, or a German, make three changes....

Change #1. Insert this line:

```
DECIMAL-POINT IS COMMA.
```

Put that line in a SPECIAL-NAMES paragraph, at the end of the ENVIRONMENT DIVISION's CONFIGURATION SECTION:

```
CONFIGURATION SECTION.
SOURCE-COMPUTER.
        The computer's name.
OBJECT-COMPUTER.
        The computer's name again.
SPECIAL-NAMES.
        DECIMAL-POINT IS COMMA.
```

Change #2. Type all numbers in French-Italian-German notation. So instead of typing—

```
MOVE 5243794.95 TO K
```

type:

```
MOVE 5243794,95 TO K
```

Change #3. Use French-Italian-German notation in pictures for *edited* numbers.

For a Frenchman or an Italian: `K PIC Z.ZZZ.ZZZ,ZZ`
For a German: `K PIC ZBZZZBZZZ,ZZ`

Those are the only changes Europeans make. They still put a period at the end of every sentence, and still use English COBOL words such as MOVE and DISPLAY.

American COBOL: `DISPLAY "HELLO, STUPID".`
French COBOL: `DISPLAY "BONJOUR, BETE".`
German COBOL: `DISPLAY "GUTEN TAG, DUMMKOPF".`

CHAPTER 7

ALTERNATIVE LANGUAGES

LANGUAGE SURVEY

EIGHT CATEGORIES

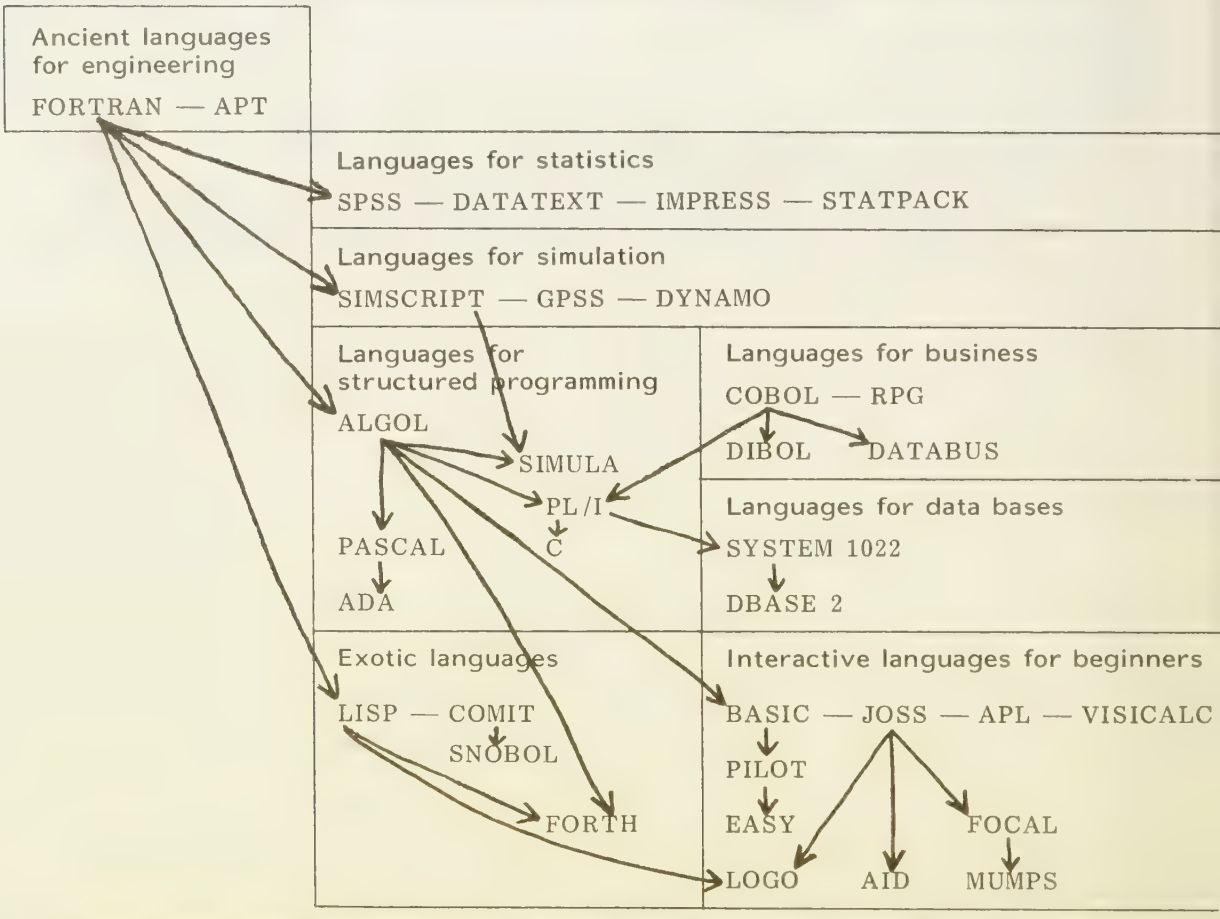
Volumes 1 and 2 explained how to program the computer by using BASIC. Volume 7 explained how to program by using FORTRAN, PASCAL, and COBOL. Volume 5 praised VISICALC (for financial planning) and DBASE 2 (for information retrieval). Those methods of communicating with the computer— BASIC, FORTRAN, PASCAL, COBOL, VISICALC, and DBASE 2— are called *high-level languages*.

In addition to those high-level languages, researchers have invented thousands of others. Most high-level languages fall into eight categories:

Category	The most popular examples
ancient languages for engineering	FORTRAN & APT
languages for statistics	SPSS & STATPACK
languages for simulation	GPSS & DYNAMO
languages for business	COBOL & RPG
languages for structured programming	PASCAL & PL/I
languages for data bases	DBASE 2 & SYSTEM 1022
exotic languages	LISP & FORTH
interactive languages for beginners	BASIC & VISICALC

FAMILY TREE

This "family tree" shows how the popular high-level languages relate to each other:



Arrows point from the parents to the children; for example; FORTRAN is the parent of SPSS, SIMSCRIPT, ALGOL, and LISP. In other words, the researchers who invented SPSS, SIMSCRIPT, ALGOL, and LISP all knew FORTRAN, and copied more features from FORTRAN than from any other major language.

A horizontal dash connects languages that are "married". For example, FORTRAN is married to APT. That means FORTRAN is similar to APT, though neither language is the parent of the other.

After a language is born, it "grows up", by incorporating features from other languages. For example, BASIC is the child of ALGOL and eventually grew up by incorporating features from many other languages, especially PL/I.

A language's life can be full of turmoil. For example, PL/I (which came from ALGOL) was originally called "FORTRAN 6", then "NPL", and then finally "PL/I"; and today, several conflicting versions of PL/I abound, using names such as "PL/C", "PL/M", and "SP/k". Another example of turmoil is SNOBOL, which was strongly influenced by COMIT but went through several conflicting versions: "SCL", then "SCL7", then "SEXI", and finally "SNOBOL".

Let's look at all those languages more closely....

ANCIENT LANGUAGES FOR ENGINEERING

Of all the languages used today, FORTRAN and APT are the oldest. And of all the languages ever invented, FORTRAN has the most descendants: FORTRAN gave birth to SPSS, SIMSCRIPT, ALGOL, and LISP, and those languages in turn gave birth to many others. APT is used in factories: it lets you make the computer tell other machines how to cut metal. Today, FORTRAN and APT seem quaint, but many programmers still use them. FORTRAN is still the most popular language for making large computers do scientific calculations. That's because large computers can handle FORTRAN more quickly than any other language.

LANGUAGES FOR STATISTICS

The most popular language for statistics is SPSS. It contains commands for handling every aspect of statistics: averages, percentiles, crosstabs, chi-square, regression analysis, and all the other gobbledygook that statisticians love. Many colleges require all psychology and sociology majors to write programs in SPSS. Unfortunately, SPSS is ugly, and the computer handles it slowly. If you want a simpler and pleasanter statistical language, choose DATATEXT or IMPRESS or STATPACK instead. But most psychologists and sociologists still use SPSS, because it contains the greatest variety of commands. Today, you can buy SPSS for practically every maxicomputer and for most minicomputers also. Only a few brands of computers can handle DATATEXT, IMPRESS, or STATPACK.

LANGUAGES FOR SIMULATION

To analyze society, make the computer imitate thousands of people interacting with each other. To make the computer imitate those thousands of people, use SIMSCRIPT, GPSS, or DYNAMO. Use GPSS to make the computer imitate people who wait in lines (at supermarkets and at barbershops and at loading docks). Use DYNAMO to make the computer predict the future (of your industry or your overpopulated city or your polluted world). Use SIMSCRIPT for weirder interactions that are too complicated for GPSS or DYNAMO.

LANGUAGES FOR BUSINESS

The most popular business language is COBOL. To write a COBOL program, use an outline and just "fill in the blanks". An even easier language is RPG, where you just "check off" what you want, by putting check marks on a four-page form. COBOL is the most popular language for IBM's *maxi*computers; RPG is the most popular language for IBM's *mini*computers. DIBOL and DATABUS are crummy stripped-down versions of COBOL: DIBOL runs only on PDP-8 and PDP-11 minicomputers; DATABUS runs only on minicomputers manufactured by Datapoint.

LANGUAGES FOR STRUCTURED PROGRAMMING

Around 1960, an international committee of mathematicians invented ALGOL. It's a very beautiful language— much more beautiful than its predecessor, FORTRAN. Many researchers tried to improve ALGOL further: those later improvements were called PASCAL, ADA, SIMULA, PL/I, and C. All those languages are "elegant". They're all praised by theoreticians in universities. They arose because of politics: PASCAL was invented by a Swiss member of the ALGOL committee because he hated the rest of the committee; ADA was invented by a French company to help Americans develop more powerful missiles to bomb Russia; SIMULA was a Norwegian attempt to combine ALGOL with SIMSCRIPT; PL/I (a combination of ALGOL, COBOL, and FORTRAN) was invented by IBM, in an unsuccessful attempt to wipe ALGOL and COBOL and FORTRAN off the face of the earth; C (a stripped-down version of PL/I) was invented by Bell Telephone, because the phone company couldn't afford to buy IBM's PL/I computers. None of those languages is widely used in the commercial world yet— although PASCAL is starting to become popular on CDC and PDP and Apple computers, and C is starting to become popular on PDP-11 computers and some microcomputers.

LANGUAGES FOR DATA BASES

To make the computer manipulate lots of data on a disk, buy a data-base program (such as PFS or DB Master) or write your *own* program. To write your own program easily, use a data-base language, such as SYSTEM 1022 or DBASE 2. The SYSTEM 1022 works only on PDP-10 and PDP-20 computers; DBASE 2 works only on microcomputers that use the CP/M operating system. Although many other data-base languages exist also, SYSTEM 1022 and DBASE 2 are the most pleasant.

EXOTIC LANGUAGES

If you're weird, you'll love the exotic languages: LISP, COMIT, SNOBOL, and FORTH. Those languages all have a strange inhuman beauty that appeals to us masochists. Use LISP to make the computer become a robot; use COMIT to make the computer understand English; use SNOBOL to make the computer do abstract algebra; and use FORTH to invent new languages for microcomputers.

INTERACTIVE LANGUAGES FOR BEGINNERS

BASIC, JOSS, APL, and VISICALC are all easy languages, for beginners. JOSS is the easiest; BASIC is the most popular; APL is the most sophisticated; VISICALC is the best for accountants. Those languages led to others: PILOT was an attempt to invent a language even easier than BASIC and JOSS; LOGO is a "kiddie" version of LISP that's been JOSSified; and the twinkle-toes trio (AID, FOCAL, and MUMPS) are just slight variations of JOSS.

WHICH LANGUAGE IS THE BEST?

All those languages seem to be leading to the "ultimate language"— which I've been developing. My language is called EASY. It's even easier to learn than PILOT, yet includes the best features of all the other languages also. It's perfect.

People often ask how I memorize so many languages. The answer is simple: to memorize a language, I just notice how it's worse than EASY and memorize its mistakes.

EASY isn't quite finished yet. If you help finish it, you'll win both fame and fortune: I'm giving out prizes. I'm putting the language in the public domain: I don't want to profit from EASY, because EASY isn't for me; it's for *you*.

THE TOP 22 LANGUAGES

Of the 35 languages in the family tree, 22 are particularly important. Here they are, in chronological order:

Name of language	What the name stands for	Original application	Where first version was invented	Name of newest version
APT	Automatically Programmed Tools	cutting metal	MIT, 1952-1957	APT 77
FORTRAN	FORmula TRANslating	sciences	IBM, 1954-1957	FORTRAN 77
ALGOL	ALGOrithmic Language	sciences	international, 1957-1958	ALGOL W, BALGOL, ALGOL 68
DYNAMO	DYNAmic MODels	simulation	MIT, 1959	DYNAMO 3
RPG	Report Program Generator	business	IBM, 1959	RPG 3
LISP	LISt Processing	artificial intelligence	MIT, 1959-1960	MACLISP, LISP-machine LISP
COBOL	COmmon Business-Oriented Language	business	Defense Dept., 1959-1960	COBOL 74
GPSS	General-Purpose Systems Simulator	simulation	IBM, 1961	GPSS 5
SNOBOL	StriNg-Oriented symBOlic Language	string processing	Bell Telephone Labs, 1962-1963	SNOBOL 4B
BASIC	Beginner's All-purpose Symbolic Instruction Code	sciences	Dartmouth College, 1963-1964	SBASIC, BASIC-PLUS-2, Microsoft BASIC
APL	A Programming Language	sciences	Harvard & IBM, 1956-1966	APLSV
PL/I	Programming Language One	general	IBM, 1963-1966	PL/I Optimizer, PL/C, ANSI PL/I
SPSS	Statistical Package for the Social Sciences	statistics	Stanford Univ., 1965-1967	SPSS 2nd edition
LOGO	LOGO	general	Bolt Beranek Newman, 1967	Apple LOGO
FORTH	FOuRTH-generation language	business & astronomy	Stanford Univ. & Mohasco, 1963-1968	POLYFORTH, FIG-FORTH, MMS FORTH
PILOT	Programmed Inquiry, Learning, Or Teaching	tutoring kids	U. of California at San Fran., 1968	Atari PILOT
PASCAL	Blaise PASCAL	general	Switzerland, 1968-1970	UCSD PASCAL
C	C	systems progr'ing	Bell Telephone Labs, 1970-1977	PDP-11 C
VISICALC	VISIble CALCulation	financial forecasting	Software Arts, 1978	VISICALC 3
ADA	ADA Lovelace	military equipment	France, 1977-1980	ADA final vers'n
DBASE 2	Data BASE 2	information retrieval	Ashton-Tate, 1978-1980	DBASE 2
EASY	EASY	general	Secret Guide, 1972-1982	EASY

The main inventors were:

Inventor	Languages
IBM	FORTRAN, RPG, GPSS, APL, PL/I
MIT	APT, DYNAMO, LISP
Bell Telephone Labs	SNOBOL, C
Stanford University	SPSS, FORTH

All the languages were invented in the United States, except for ALGOL, PASCAL, and ADA.

The "dead era" lasted from 1969 to 1979: during that 11 year stretch, only three languages were finished: PASCAL, C, and VISICALC.

The first thirteen languages (up through SPSS) are the "classics". They've withstood the test of time. Everybody agrees that those languages made significant contributions to our culture.

The remaining nine languages (beginning with LOGO) are new and controversial; the jury isn't in yet. What's *your* verdict? Do you think these new languages are wonderful or terrible?

Each of those 22 languages— even the oldest— is still used today. And each language teaches you a different philosophy of programming, a different way to look at life. To get the *whole* truth, you must look at both the *old* truths and the *new*. If you learn the old without the new— or the new without the old— you'll be only *half*-wise. Study the languages— *all* of them!

To help you along, the next page shows a multilingual dictionary for the "mainstream" languages. And here's how to add 2 and 2:

BASIC
```
10 PRINT 2+2
```

FORTRAN
```
N=2+2
PRINT *, N
END
```

PASCAL
```
PROGRAM HARRY;
BEGIN;
WRITELN(2+2);
END.
```

COBOL
```
IDENTIFICATION DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01      N PIC 9.
PROCEDURE DIVISION.
MAIN-ROUTINE.
        ADD 2 2 GIVING N.
        DISPLAY N.
        STOP RUN.
```

ALGOL
```
BEGIN   PRINT(2+2);
END
```

ADA
```
PROCEDURE HARRY IS
BEGIN
        PUT(2+2);
END;
```

PL/I
```
HARRY:  PROCEDURE OPTIONS(MAIN);
        PUT LIST(2+2);
        END;
```

C
```
MAIN()
{
        PRINTF("%D\N",2+2);
}
```

SNOBOL
```
        OUTPUT = 2 + 2
END
```

PILOT
```
C:#K=2
C:#N=#K+2
T:#N
```

APL
```
2+2
```

LOGO
```
PRINT 2+2
```

LISP
```
(PLUS 2 2)
```

FORTH
```
2 2 + .
```

VISICALC
```
+2+2
```

EASY
```
SAY 2+2
```

The next chapter looks at each of the 22 languages in chronological order, beginning with the classics.

BASIC	DIM X(4)	FOR I = 5 TO 17		GOSUB 1000	GO TO 50
FORTRAN	DIMENSION X(4)	DO 10 I=5,17		CALL JOE	GO TO 50
PASCAL	X: ARRAY [1..4] OF REAL	FOR I := 5 TO 17 DO		JOE	GOTO 50
COBOL	X OCCURS 4 TIMES	PERFORM SAM VARYING I FROM 5 BY 1 UNTIL I > 17		PERFORM JOE	GO TO GAIL
ALGOL	REAL ARRAY X[1:4]	FOR I := 5 STEP 1 UNTIL 17 DO		JOE	GO TO GAIL
ADA	X: ARRAY(1..4) OF FLOAT	FOR I IN 5..17 LOOP		JOE	GO TO GAIL
PL/I	DECLARE X(4)	DO I = 5 TO 17		CALL JOE	GO TO GAIL
C	FLOAT X[4]	FOR (I=5; I<=17; ++I)		X=JOE()	GOTO GAIL
SNOBOL	X = ARRAY(4)	*not available*		JOE()	:(GAIL)
PILOT	DIM:#X(4)	*not available*		U:JOE	J:*GAIL
APL	*not needed*	*not available*		JOE	→50
LOGO	DEFAR "X 4 1	*not available*		JOE	GO "GAIL
LISP	(ARRAY ((X (4) LIST)))	*not available*		(JOE)	(GO GAIL)

BASIC	IF X=4.3 THEN GO TO 100	INPUT K	J=K+2	PRINT K	REM DUMB STUFF
FORTRAN	IF (X .EQ. 4.3) GO TO 100	READ *, K	J=K+2	PRINT *, K	C DUMB STUFF
PASCAL	IF X=4.3 THEN GOTO 100	READ(K)	J:=K+2	WRITELN(K)	{DUMB STUFF}
COBOL	IF X = 4.3 GO TO KATE	ACCEPT K	COMPUTE J = K + 2	DISPLAY K	*DUMB STUFF
ALGOL	IF X=4.3 THEN GO TO KATE	READ(K)	J:=K+2	PRINT(K)	COMMENT DUMB STUFF
ADA	IF X=4.3 THEN GO TO KATE; END IF	GET(K)	J:=K+2	PUT(K)	--DUMB STUFF
PL/I	IF X=4.3 THEN GO TO KATE	LET LIST(K)	J=K+2	PUT LIST(K)	/*DUMB STUFF*/
C	IF (X==4.3) GOTO KATE	SCANF("%D",&K)	J=K+2	PRINTF("%D\N",K)	/*DUMB STUFF*/
SNOBOL	EQ(X,4.3) :S(KATE)	K = INPUT	J = K + 2	OUTPUT = K	*DUMB STUFF
PILOT	*not available*	A:#K	C:#J=#K+2	T:#K	R:DUMB STUFF
APL	→(X=4.3)/100	K←□	J←K+2	K	⍝DUMB STUFF
LOGO	IF :X=4.3 GO 100	MAKE "K TYPEIN	MAKE "J :K+2	PRINT :K	!DUMB STUFF
LISP	(COND ((EQUAL X 4.3) (GO KATE)))	(SETQ K (READ))	(SETQ J (PLUS K ?))	K	;DUMB STUFF

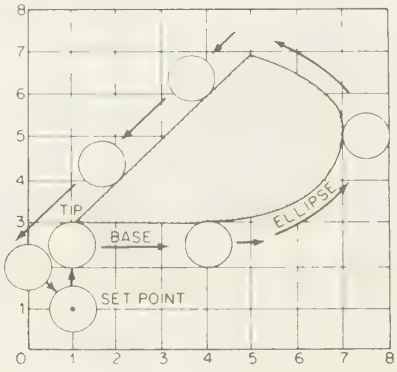
APT

WHY USE APT?

If you use APT, the computer will help you cut metal. Type an APT program that says how you want the metal cut. When you run the program, the computer will create a special instruction tape. Feed the instruction tape into a metal-cutting machine, and the machine will cut metal as you said.

EXAMPLE

Let's write an APT program that makes the machine cut out the shaded area:



We'll make the machine move the cutter where the circles are.

Here's the program:

```
CUTTER/1                    Use a cutter whose diameter is 1".
TOLER/.005                  The tolerance of the cut is .005".
FEDRAT/80                   Use a feedrate of 80" per minute.
HEAD/1                      Use head 1.
MODE/1                      Operate the tool in mode 1.
SPINDL/2400                 Turn the spindle on, at 2400 rpm.
COOLNT/FLOOD                Turn the coolant on, at flood setting.
PT1=POINT/4,5               PT1 = the point whose coordinates are (4,5).
FROM/(SETPT=POINT/1,1)      SETPT = the point (1,1). Start the tool from SETPT.
INDIRP/(TIP=POINT/1,3)      TIP = (1,3). Aim the tool in the direction of the point TIP.
BASE=LINE/TIP, AT ANGL, 0   BASE = the line that goes through TIP and slants 0 degrees.
GOTO/BASE                      Make the tool go to BASE.
TL RGT, GO RGT/BASE            With the tool on the right, go right along BASE.
GO FWD/(ELLIPS/CENTER, PT1, 3,2,0)  Go forward along the ellipse whose center is PT1,
                                    semi-major axis is 3", semi-minor axis is 2",
                                    and major axis slants 0 degrees.
GO LFT/(LINE/2,4,1,3,), PAST, BASE  Go left along the line that joins (2,4) and (1,3),
                                    until you get past BASE.
GOTO/SETPT                  Make the tool go to SETPT.
COOLNT/OFF                  Turn the coolant off.
SPINDL/OFF                  Turn the spindle off.
END                         End use of the machine.
FINI                        The program is finished.
```

FORTRAN

HOW FORTRAN DEVELOPED

In 1954 an IBM committee said it was planning a language called FORTRAN. The plans resembled today's FORTRAN but did not allow long variable names, subroutines, long function definitions, double precision, complex numbers, or apostrophes.

According to the plans, a variable's name had to be very short—only two letters. A function's definition had to fit on a single line. Instead of using apostrophes and writing 'PLEASE KISS ME', the programmer had to write 14HPLEASE KISS ME; the 14H meant a 14-character string.

The plans did *not* require a DO loop to be immediately under the DO statement. The loop could be elsewhere, and the DO statement would say where.

There was no END statement. The programmer could warn the computer how frequently a variable would appear in the program, by saying FREQUENCY. Besides comparing numbers, the IF statement could say IF ACCUMULATOR OVERFLOW or IF QUOTIENT OVERFLOW.

The first working version of FORTRAN (1957) allowed longer variable names—up to 6 characters—and put the DO loop under the DO statement. FORTRAN II (1958) allowed subroutines and long function definitions, and put an END statement at the end of the program. IBM experimented with a FORTRAN III, but never released it to the public. FORTRAN IV (1962) allowed double precision and complex numbers, and abolished the statements about FREQUENCY and OVERFLOW. Apostrophes weren't allowed until later.

The original plans said you'd be able to add an integer to a real. That didn't work in FORTRAN I, FORTRAN II, and FORTRAN IV, but it works today.

The original plans said an IF statement would compare any two numbers. FORTRAN I and FORTRAN II required one of the numbers to be zero, but FORTRAN IV removed the restriction.

Before FORTRAN was invented, programming was so difficult that only a few specialists could do it. IBM waged a campaign to convince everyone that FORTRAN was easier. IBM succeeded: people who knew little about computers wrote their own FORTRAN programs.

Other manufacturers imitated IBM's FORTRAN, but made some changes. In 1966, the American National Standards Institute (ANSI) created a single version of FORTRAN IV for all manufacturers. Now ANSI has completed an improvement, called ANSI Standard FORTRAN 77. ("ANSI" is pronounced "an see".)

IS FORTRAN GOOD?

FORTRAN is the favorite language among scientists, because it allows four kinds of numbers (integers, reals, double precision, and complex), allows many formats for input and output, lets you decide which variables in the main routine relate to which variables in the subroutine, and lets the computer run loops more quickly than other languages.

But FORTRAN is not perfect. IBM's first FORTRAN manual (1956) said, "FORTRAN is intended to be capable of expressing any problem of numerical computation. However, for problems that have a logical rather than a numerical meaning, FORTRAN is less satisfactory." In other words, FORTRAN handles numbers well, but handles string variables awkwardly. FORTRAN 77 is an improvement but still doesn't bring FORTRAN's string capabilities up to the level of BASIC, PL/I, and SNOBOL.

Jean Sammet, the woman who's the world expert on the history of computer languages, and whose books and papers analyze more than 200 languages, says, "Because FORTRAN was designed so early, better ways have been found to do almost everything that is currently in FORTRAN." In other words, FORTRAN is outdated. John Kemeny, who invented BASIC, says, "FORTRAN is as outdated as Latin." New languages, such as BASIC, ALGOL, and APL, are easier to learn than FORTRAN. PL/I is more flexible. But scientists still use FORTRAN, because those newer languages have disadvantages that offset their advantages.

ALGOL

SIMPLE PROGRAMS

From 1958 to 1962, the top computer organizations in America and Europe tried to invent a perfect language. They called it "ALGOL". It's simple and beautiful.

Here's how to write ALGOL programs on a PDP-10 computer. (If you're using an IBM computer, make the changes listed at the end of the chapter.)

Here's an ALGOL program:

```
BEGIN   PRINT(7+5-1);
        PRINT(52-(10+10));
        PRINT(-8+1);
END
```

An ALGOL program must begin with the word BEGIN, and end with the word END. Each statement must end with a semicolon. (Exception: in the program above, you can omit the semicolon at the end of line 3, since END comes right after it.) The program makes the computer print:

```
          11          32          -7
```

That's the traditional way to type an ALGOL program. You can put several statements on the same line and not indent—

```
BEGIN PRINT(7+5-1); PRINT(52-(10+10)); PRINT(-8+1); END
```

but the traditional style is prettier.

As in FORTRAN, you can use integers and reals. For reals, put digits after the decimal point:

INTEGER	REAL	WRONG
5	5.0	5.

For powers of ten, use the symbol &:

FORTRAN	ALGOL
4.3E6	4.3&6

You can use these operations:

OPERATION	SYMBOL	TYPE OF ANSWER
addition	x+y	real, unless x and y are both integers
subtraction	x-y	real, unless x and y are both integers
multiplication	x*y	real, unless x and y are both integers
division	x/y	always real
exponents	x↑y	real, unless x and y are both integers and y is non-negative

Here are examples:

```
5+3.0 is 8.0
5+3 is 8
3/4 is .75        FORTRAN would give 0 instead
16/2 is 8.0       FORTRAN would give 8 instead
2↑(-3) is .125    FORTRAN would give 0 instead
```

You can use these functions:

FUNCTION	SYMBOL
square root	SQRT(x)
sine	SIN(x)
cosine	COS(x)
arctangent	ARCTAN(x)
logarithm	LN(x)
e^x	EXP(x)

The x can be either an integer or a real. But even if x is an integer, the answer is real. For example, SQRT(9) is 3.0, and SQRT(2) is 1.4142135. *In FORTRAN, you'd have to say SQRT(9.0) and SQRT(2.0) instead.*

For strings, say WRITE instead of PRINT:

```
BEGIN   WRITE("HOT");
        WRITE("DOG");
        NEWLINE;
        WRITE("MAN");
END
```

The computer will write HOT, then DOG, then on a new line write MAN:

```
HOTDOG
MAN
```

The computer will ignore any statement that begins with the word COMMENT.

VARIABLES

The name of a variable can be short (like X and N and COST2) or long—up to 64 letters and digits. It must begin with a letter.

Tell the computer which variables are integers and which are reals:

```
BEGIN   REAL X; INTEGER Y,Z;
        X:=SQRT(2);
        Y:=5+5;
        Z:=4↑2;
        PRINT(X+Y+Z);
END
```

The first line says X is real, and Y and Z are integers. X is the real number 1.4142136, Y is the integer 10, and Z is the integer 16. The computer will print the sum, 27.414213, in scientific notation.

If you don't say whether X, Y, and Z are integers or reals, the computer will gripe. Saying whether a variable is an integer or a real is called a *declaration.* The declarations must come immediately after the word BEGIN.

If you say J is real and say J:=5, the J will be 5.0. If you say K is an integer and say K:=2.9, the computer will round 2.9 to the nearest integer, so K will be 3. *FORTRAN would ignore the .9 and make K be 2.*

```
BEGIN   REAL X;
        WRITE("TYPE A NUMBER");
        NEWLINE;
        READ(X);
        WRITE("THE SQUARE OF YOUR NUMBER IS"); PRINT(X↑2);
END
```

The computer will write TYPE A NUMBER. It will read the number you type and will print its square.

In a program that outputs and then READs, say NEWLINE in between.

```
BEGIN   WRITE("GOOD ");
        GO TO MOMMY;
        WRITE("BAD ");
MOMMY:  WRITE("APPLE ");
        WRITE("PIE");
END
```

The computer will write:
GOOD APPLE PIE
In that program, MOMMY is the fourth statement's *label*.

```
BEGIN   INTEGER I;
        FOR I:= 2, 7, 10 DO PRINT(I↑2);
        WRITE("HELLO");
END
```

The computer will print 4, 49, and 100. Then it will write HELLO.

STATEMENT	MEANING
FOR I:=10 STEP 1 UNTIL 15 DO PRINT(I↑2);	Print the squares of 10, 11, 12, 13, 14, and 15.
FOR I:=2, 11, 7 STEP -.3 UNTIL 5 DO PRINT(I↑2);	Print the squares of 2, 11, 7, 6.7, 6.4, 6.1, 5.8, 5.5, and 5.2.
IF X<4 THEN WRITE("CAT");	If X<4, write CAT.
IF X<4 THEN WRITE("CAT") ELSE PRINT(8+9);	If X<4, write CAT. If not, print 17.
Y:=IF X<4 THEN 8 ELSE 9;	Let Y=8, if X<4. If not, let Y=9.
Y:=N+(IF X<4 THEN 8 ELSE 9);	Let Y=N+8, if X<4. If not, let Y=N+9.
IF X<4 THEN GO TO MOMMY;	If X<4, go to MOMMY.
GO TO IF X<4 THEN MOMMY ELSE DADDY;	Go to MOMMY, if X<4. If not, go to DADDY.

This statement is so long it takes two lines:

```
        IF P<.03 THEN WRITE("HICCUP")
        ELSE IF X<4 THEN WRITE("CAT") ELSE PRINT(8+9);
```

Since those lines contain only one semicolon, they form only one statement, not two. It means, "If P is less than .03, write HICCUP; if not less than .03, do the following: if X is less than 4, write CAT; if not less than 4, print 17."

You can use these conditions:

CONDITION	MEANING
IF X=4	if X equals 4
IF X<=4	if X is less than or equal to 4
IF X>=4	if X is greater than or equal to 4
IF X#4	if X is not equal to 4
IF X>4 AND X<10 OR X>20 AND X<30	if X is greater than 4 and less than 10, or greater than 20 and less than 30

This program's simple:

```
BEGIN   WRITE("BILL ");
        WRITE("KICKS ");
        WRITE("MARY");
END
```

The computer will print:
BILL KICKS MARY

This program does the same thing:

```
BEGIN   PROCEDURE K; WRITE("KICKS ");
        WRITE("BILL ");
        K;
        WRITE("MARY");
END
```

The top line defines K to be WRITE("KICKS "). The rest of the program says to write BILL, then do K, then write MARY.

```
BEGIN   PROCEDURE T; WRITE("TICKLISH ");
        WRITE("I FEEL ");
        T;
        T;
        T;
END
```
The computer will write:
I FEEL TICKLISH TICKLISH TICKLISH

```
BEGIN   INTEGER I;
        PROCEDURE DOUBLE; I:=2*I;
        I:=7;
        DOUBLE;
        DOUBLE;
        PRINT(I);
END
```
The second line defines DOUBLE to be the procedure of doubling I. The computer will let I be 7, then do a DOUBLE, then another DOUBLE, and finally print 28.

Procedures should be defined at the beginning, immediately after BEGIN and declarations such as INTEGER I.

```
BEGIN   PROCEDURE P; GO TO MOM;
        WRITE("MARY ");
        P;
        WRITE("HATES ");
MOM:    WRITE("LIES");
END
```
The top line defines P to be GO TO MOM. The computer will write MARY, then do P, which makes it go to MOM and write LIES:
MARY LIES

COMPOUND STATEMENTS

Here's a simple statement:

```
WRITE("MEAT");
```

This *compound statement* does the same thing:

```
BEGIN
        WRITE("ME");
        WRITE("AT");
END;
```

Like a program, a compound statement begins with BEGIN and ends with END. Like all statements, a compound statement ends with a semicolon.

LINE	EQUIVALENT LINES

`FOR I:=1 STEP 1 UNTIL 10 DO WRITE("MEAT");`	`FOR I:=1 STEP 1 UNTIL 10 DO`
	`BEGIN`
	`  WRITE("ME");`
	`  WRITE("AT");`
	`END;`

`IF X<4 THEN WRITE("MEAT");`	`IF X<4 THEN`
	`BEGIN`
	`  WRITE("ME");`
	`  WRITE("AT");`
	`END;`

`IF X<4 THEN WRITE("MEAT") ELSE PRINT(5+2);`	`IF X<4 THEN`
	`BEGIN`
	`  WRITE("ME");`
	`  WRITE("AT");`
	`END` *no semicolon here, since the IF*
	`ELSE PRINT(5+2);` *statement isn't done*

`PROCEDURE P; WRITE("MEAT");`	`PROCEDURE P;`
	`BEGIN`
	`  WRITE("ME");`
	`  WRITE("AT");`
	`END;`

This program computes $\sqrt{1}+\sqrt{2}+\sqrt{3}+\sqrt{4}+\sqrt{5}+\sqrt{6}+\sqrt{7}+\sqrt{8}+\sqrt{9}+\sqrt{10}$, and also $\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+\frac{1}{9}+\frac{1}{10}$:

```
BEGIN   INTEGER I; REAL SUMSQUARES,SUMRECIPROCALS;
        SUMSQUARES:=0.0; SUMRECIPROCALS:=0.0;
        FOR I:=1 STEP 1 UNTIL 10 DO
        BEGIN
                SUMSQUARES:=SUMSQUARES+SQRT(I);
                SUMRECIPROCALS:=SUMRECIPROCALS+1/I;
        END;
        PRINT(SUMSQUARES); PRINT(SUMRECIPROCALS);
END
```

FORTRAN would require FLOAT(I) instead of I

FORTRAN would require 1.0 instead of 1

This program does the same:

```
BEGIN   INTEGER I; REAL SUMSQUARES,SUMRECIPROCALS;
        PROCEDURE BIGGERSUMS;
        BEGIN
                SUMSQUARES:=SUMSQUARES+SQRT(I);
                SUMRECIPROCALS:=SUMRECIPROCALS+1/I;
        END;
        SUMSQUARES:=0.0; SUMRECIPROCALS:=0.0;
        FOR I:=1 STEP 1 UNTIL 10 DO BIGGERSUMS;
        PRINT(SUMSQUARES); PRINT(SUMRECIPROCALS);
END
```

definition of BIGGERSUMS

main routine

```
This program is tricky:
BEGIN   INTEGER I;
        I:=5;
        BEGIN   INTEGER I;
                I:=7;
                PRINT(I);
        END;
        PRINT(I*2);
END
```

In that program, the computer sets I equal to 5. Then the computer does the compound statement:

```
        BEGIN   INTEGER I;
                I:=7;
                PRINT(I);
        END;
```

Since the compound statement begins by saying INTEGER I, the computer creates a temporary value of I, just for the compound statement. That value is 7, and the computer prints it. When the computer finishes the compound statement, it forgets the temporary value of I, and returns to the original value, 5. The program's last statement makes the computer print 25.

The temporary value of I, which is 7, is called the *local value*. The general value of I, which is 5, is the *global value*. A compound statement that creates a temporary value is called a *block*.

Suppose you're writing a long program that uses a real variable S in only one compound statement. Say REAL S at the beginning of the compound statement, instead of at the beginning of the program. Here's why:

1. The program will be easier for a human to read, since the note REAL S will be closer to where S is used.
2. The program will hog less computer memory, since S will use memory only when the compound statement is being executed.
3. The program will be easier to expand, since your temporary S won't conflict with any other S you create later.

```
BEGIN   REAL ARRAY A[4:7];
        A[4]:=1.3;
        A[5]:=.1;
        A[6]:=.2;
        A[7]:=2.0;
        PRINT(A[4]+A[5]+A[6]+A[7]);
END
```

The top line says A[4], A[5], A[6], and A[7] are real numbers. The computer will print the sum, 3.6.

Instead of saying A[4:7], you can say A[L:H], and let L and H be input:

```
BEGIN   INTEGER L,H;
        WRITE("WHAT IS L?");
        NEWLINE;
        READ(L);
        WRITE("WHAT IS H?");
        NEWLINE;
        READ(H);
        BEGIN   REAL ARRAY A[L:H]; INTEGER I;
                WRITE("WHAT ARE THE VALUES OF A?");
                NEWLINE;
                FOR I:=L STEP 1 UNTIL H DO READ(A[I]);
                etc.
        END;
END
```

ALGOL's ability to let you input a dimension is called *dynamic storage allocation*. On most computers, BASIC and FORTRAN don't allow you to input a dimension; the dimension must be a constant; the storage allocation is *static*.

This statement says A will have two dimensions:

```
        REAL ARRAY A[4:7, -8:10];
```

The first subscript will range from 4 to 7. The second will range from -8 to 10. The first value of A will be A[4,-8]; the last will be A[7,10].

Here's how to read a table:

```
BEGIN   INTEGER R,C;
        WRITE("HOW MANY ROWS WILL THE TABLE HAVE?");
        NEWLINE;
        READ(R);
        WRITE("HOW MANY COLUMNS?");
        NEWLINE;
        READ(C);
        BEGIN   REAL ARRAY A[1:R, 1:C]; INTEGER I,J;
                WRITE("WHAT ARE THE ENTRIES OF THE TABLE?");
                FOR I:=1 STEP 1 UNTIL R DO
                FOR J:=1 STEP 1 UNTIL C DO READ(A[I,J]);
                etc.
        END;
END
```

Instead of saying REAL ARRAY, you can say ARRAY. The computer assumes the array is real, unless you specify otherwise.

Instead of writing——

```
        INTEGER ARRAY A[1:50]; INTEGER ARRAY B[1:50]; INTEGER ARRAY C[1:50];
```

you can write:

```
        INTEGER ARRAY A,B,C[1:50];
```

HOW ALGOL DEVELOPED

In 1955, the Gesellschaft für Angewandte Mathematik und Mechanik (German Association for Applied Mathematics and Mechanics) created a committee on programming, which later created a subcommittee that began inventing a computer language. The subcommittee decided the language should use English terms instead of German, since English was the international language for science.

In 1957 the Germans invited Americans to join them. In 1958 other European countries joined also, and the international committee proposed a new language, called "IAL" (International Algebraic Language). Then the committee renamed the language "ALGOL 58", wrote a revision called "ALGOL 60", wrote another revision, called "ALGOL 60 (Revised)", and then disbanded. Today when programmers mention "ALGOL", they usually mean the committee's last report, ALGOL 60 (Revised).

ALGOL is more convenient than FORTRAN:

1. ALGOL's variables can have long names. FORTRAN allows only 6 characters. ALGOL allows 64 on PDP-10 computers, and even more on others.
2. Writing 2 instead of 2.0 doesn't affect ALGOL's answer. In ALGOL, 1/2 gives the same answer as 1/2.0; 2↑(-3) gives the same as 2.0↑(-3); and SQRT(2) gives the same as SQRT(2.0).
3. ALGOL's IF is flexible. It can contain ELSE, BEGIN, and END, and you can say Y:=N+(IF X<4 THEN 8 ELSE 9).
4. ALGOL's FOR is flexible. ALGOL lets you say FOR X:=3.7, Y+6.2, SQRT(Z) STEP -.3 UNTIL 5 DO. FORTRAN's DO is more restrictive; some versions of FORTRAN even insist that the DO statement must not contain reals, must not contain negatives, and must not contain arithmetic operations.
5. ALGOL allows dynamic storage. FORTRAN allows DIMENSION X(20) but not X(N). ALGOL allows ARRAY X[1:N].

Europeans began using ALGOL more than FORTRAN. The American computer association (called the Association for Computing Machinery, ACM) said all programs in its magazine would be in ALGOL.

But IBM didn't put ALGOL on its computers. Most American programmers used IBM computers and therefore couldn't use ALGOL. This created a ridiculous situation: programmers programmed in FORTRAN instead, but submitted ALGOL translations to the ACM's magazine, which published the ALGOL versions, which the magazine's readers had to translate back to FORTRAN in order to run them on IBM computers.

IBM computers eventually swept over Europe, so that even Europeans had to use FORTRAN instead of ALGOL.

Finally, the ACM gave in, and said it would publish programs in FORTRAN. But since ALGOL was prettier, everybody continued to submit ALGOL versions anyway.

IBM gave in too, and put ALGOL on its computers. But IBM's version of ALGOL is so limited and awkward that nobody takes it seriously, and even IBM itself has stopped supporting it. In 1972 Stanford University created ALGOL W, a better version that runs on IBM computers. But ALGOL W came too late: universities and businessmen had already committed themselves to FORTRAN.

From that quickie history, you might think the downfall of ALGOL was IBM's fault. But there are other sides to the story.

IBM had invested 25 man-years to develop the first version of FORTRAN. By the time the ALGOL committee finished the report on ALGOL 60 Revised, IBM had also developed FORTRAN II and FORTRAN III and made plans for FORTRAN IV. IBM was proud of its FORTRAN idea and wanted to elaborate on it.

Also, IBM realized that computers run FORTRAN programs more quickly than ALGOL

When asked why it didn't support ALGOL, IBM replied that the committee's description of ALGOL was incomplete. IBM was right: the ALGOL 60 Revised Report has three loopholes:

1. The report doesn't say what words to use for input and output, because the committee couldn't agree. So computers differ. If you want to transfer an ALGOL program from one computer to another, you must change all the input and output instructions.
2. The report uses symbols such as ÷ and ∧, which aren't on most terminals. The report underlines keywords; most terminals can't underline. Terminals must substitute other symbols for ÷ and ∧ and underlining. Here again, manufacturers differ. To transfer an ALGOL program to another manufacturer, you must change symbols.
3. Some features of ALGOL are hard to teach to a computer. Even today, no computer understands all of ALGOL. When a manufacturer says its computer "understands ALGOL", you must ask, "*Which* features of ALGOL?" In the chapter on ALGOL, I explained just the features most computers have. But even if your program uses just those features, it will run more slowly than if it were written in FORTRAN

Two other ALGOL committees were formed later. One committee developed suggestions on how to do input and output, but the suggestions have been largely ignored. The other committee tried to invent a better ALGOL; the committee wrote its preliminary report in 1968 and revised it in 1975. Called "ALGOL 68 (Revised)", the report is weird: it even requires you to spell words backwards (IF becomes FI, and DO becomes OD). The committee's decision was far from unanimous, and several members refuse to endorse the report. ALGOL 68 will probably be used even less than ALGOL 60 Revised.

ALGOL led to BASIC, PL/I, and PASCAL. Those new languages incorporate ALGOL's good features and avoid its deficiencies.

ALGOL ON IBM COMPUTERS

IBM computers use different symbols:

PDP-10 COMPUTERS USING ALGOL	IBM COMPUTERS USING ALGOL W	IBM COMPUTERS USING IBM ALGOL
7↑2	7**2	7**2
A[4]	A(4)	A(/4/)
AND	AND	&
OR	OR	|
X≠4	X¬=4	X¬=4
4.3&6	4.3'6	4.3'6
PRINT(X);	WRITEON(X);	OUTINTEGER(1,X); *or* OUTREAL(1,X);
WRITE("LOVE");	WRITEON("LOVE");	OUTSTRING(1,'('LOVE')');
NEWLINE;	WRITE();	SYSACT(1,14,1);
READ(X);	READON(X);	ININTEGER(0,X); *or* INREAL(0,X);

On IBM computers using IBM ALGOL, put apostrophes around all keywords:
'BEGIN' 'END' 'INTEGER' 'REAL' 'GOTO' 'FOR' 'STEP' 'UNTIL' 'DO'
'IF' 'THEN' 'ELSE' 'PROCEDURE' 'ARRAY'

On IBM computers using ALGOL W, you can use these abbreviations:

LINE	ABBREVIATION
WRITEON(X); WRITEON(Y); WRITEON(Z);	WRITEON(X,Y,Z);
WRITEON(X); WRITEON(Y); WRITEON(Z); WRITE();	WRITE(X,Y,Z);

DYNAMO

SYMBOLS

DYNAMO uses these symbols:

SYMBOL	MEANING
.J	a moment ago
.K	now
.JK	during the past moment
.KL	during the next moment
DT	how long "a moment" is

For example, suppose you want to explain to the computer how population depends on birth rate and death rate. If you let P be the population, BR be the birth rate, and DR be the death rate, here's what to say in DYNAMO:

```
  P.K=P.J+DT*(BR.JK-DR.JK)
```

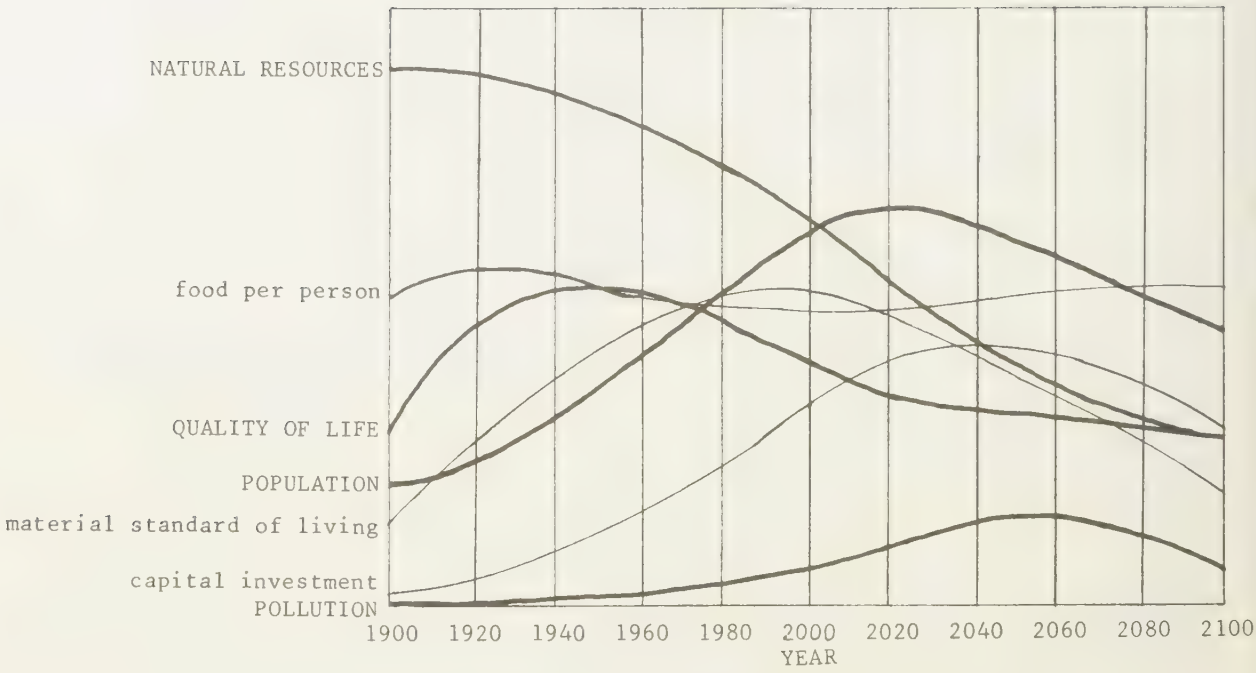
The equation says: Population now = Population before + how long "a moment" is * (Birth Rate during the past moment - Death Rate during the past moment).

WORLD DYNAMICS

The most famous DYNAMO program is the "World Dynamics Model", which Jay Forrester programmed at MIT in 1970. His program has 117 equations that describe 112 variables about our world. Here's how the program begins:

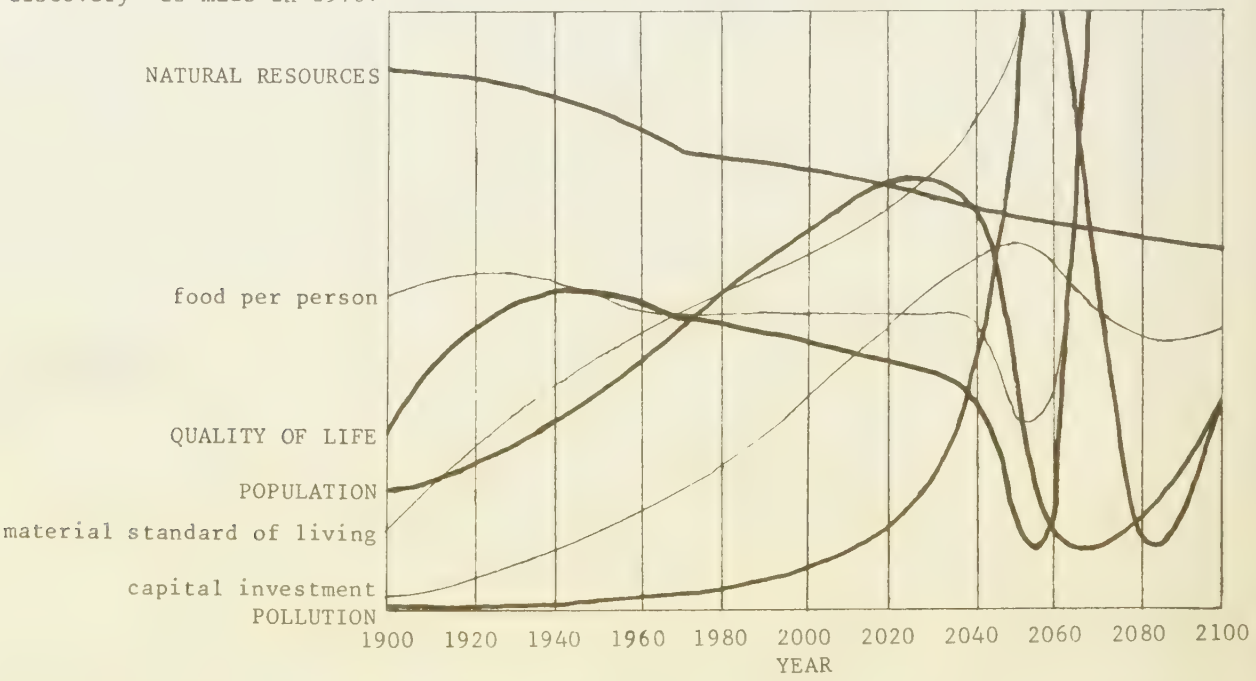
```
* WORLD DYNAMICS
L P.K=P.J+DT*(BR.JK-DR.JK)
N P=PI
C PI=1.65E9
R BR.KL=P.K*FIFGE(BRN,BRN1,SWT1,TIME.K)*BRFM.K*BRMM.K*BRCM.K*BRPM.K
etc.
```

The first line gives the program's title. The next line defines the Level of Population, in terms of Birth Rate and Death Rate. The second equation defines the iNitial Population to be PI (Population Initial); the next equation defines the Constant PI to be 1.65E9, because the world's population was 1.65 billion in 1900. The next equation says the Rate BR.KL (the Birth Rate during the next moment) is determined by the Population now and several other factors, such as the BRFM (Birth-Rate-from-Food Multiplier), the BRMM (Birth-Rate-from-Material Multiplier), the BRCM (Birth-Rate-from-Crowding Multiplier), and the BRPM (Birth-Rate-from-Pollution Multiplier). Each of those factors is defined in later equations. When you run the program, the computer automatically solves all the equations simultaneously and draws graphs that show how the population, birth rate, etc. will change during this century and the next. Here are some of the results:



The graph shows the quality of life will decrease because of overpopulation, pollution, and dwindling natural resources. Although the material standard of living will improve for a while, it too will eventually decrease, as will industrialization (capital investment).

Dwindling natural resources are the main problem. Suppose that problem is suddenly solved, by a new discovery that lets us reduce our usage of natural resources by 75%. Will our lives be better? Here's what the computer predicts, if the "new discovery" is made in 1970:



The graph of natural resources changes sharply in 1970, because of the "new discovery." As a result, people live well, so that in 2030 the population is almost [illegible] times what it was in 1970. But the large population generates a lot of pollution; in 2030, the pollution is being created faster than it can dissipate. From 2030 to 2060, a pollution crisis occurs: the pollution increases until it is 40 times as great as in 1970; most of mankind consequently dies, so that the population in 2060 is a sixth of what it was in 2040. After the crisis, the few survivors create little pollution and enjoy a very high quality of life.

Forrester tried other experiments on the computer. He tried birth control, pollution reduction, and other techniques. Each of them backfired. The graphs showed that the only way to maintain a high quality of life throughout the next century is to adopt a *combination* strategy now:
reduce natural resource usage by 75%
reduce pollution generation by 50%
reduce the birth rate by 30%
reduce capital-investment generation by 40%
reduce food production by 20%

OTHER POPULAR APPLICATIONS

DYNAMO has been applied to other problems too. It was originally designed to help you manage a company. Plug your policies about buying, selling, hiring, and firing into the program's equations; when you run the program, the computer will draw a graph showing what will happen to your company during the next few months and years. If you don't like what you see, change your policies, put the new policies into the equations, and see whether the graphs are more optimistic.

HOW DYNAMO DEVELOPED

At MIT in 1958, Richard Bennett invented a language called "SIMPLE", which stands for "Simulation of Industrial Management Problems with Lots of Equations". In 1959, Phyllis Fox and Alexander Pugh III invented the first version of DYNAMO, by improving on SIMPLE. In 1961, Jay Forrester wrote a book called *Industrial Dynamics*, which explained how DYNAMO can help you manage a company.

MIT is near Boston, whose mayor from 1960 to 1967 was John Collins. When he stepped down, he became a visiting professor at MIT. His office happened to be next to Forrester's. He asked Forrester whether DYNAMO could solve the problems of managing a city. Forrester organized a conference of urban experts and got them to turn urban problems into 330 DYNAMO equations involving 310 variables. Forrester ran the program and made the computer graph the consequences. The results were surprising.

The graphs showed that if you try to help the underemployed, by giving them low-cost housing, job-training programs, and artificially-created jobs, the city becomes better for the underemployed, so more underemployed people move to the city, so the percentage of the city that is underemployed increases, and the city is worse off than before the reforms were begun. In other words, socialist reform just backfires. Another example: free public transportation creates *more* traffic, because it encourages people to live farther from their jobs.

Instead, the graphs show that the only long-term solution to the city's problems is to knock down slums, fund new "labor-intensive export" businesses (businesses that will hire many workers, occupy little land, and produce goods that can be sold outside the city), and let the underemployed fend for themselves in this new environment. Another surprise: any city-funded housing program makes matters *worse*—regardless of whether the housing is for the underemployed, the workers, or the rich—because additional housing means less space for industry and hence fewer jobs.

If you ever become a mayor or President, use the computer's recommendations cautiously: they'll improve the cities, but only by driving the underemployed out to the suburbs, which will worsen.

In 1970 Forrester programmed the World Dynamics Model, to help "The Club of Rome", a private club of 75 people who are trying to help the world avoid ecological calamity.

RPG

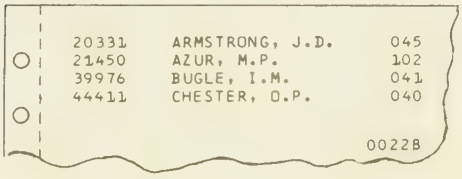
WHICH COMPUTERS PREFER RPG?

RPG is the most popular language for small IBM computers, such as the IBM System/3 and the IBM System/32.

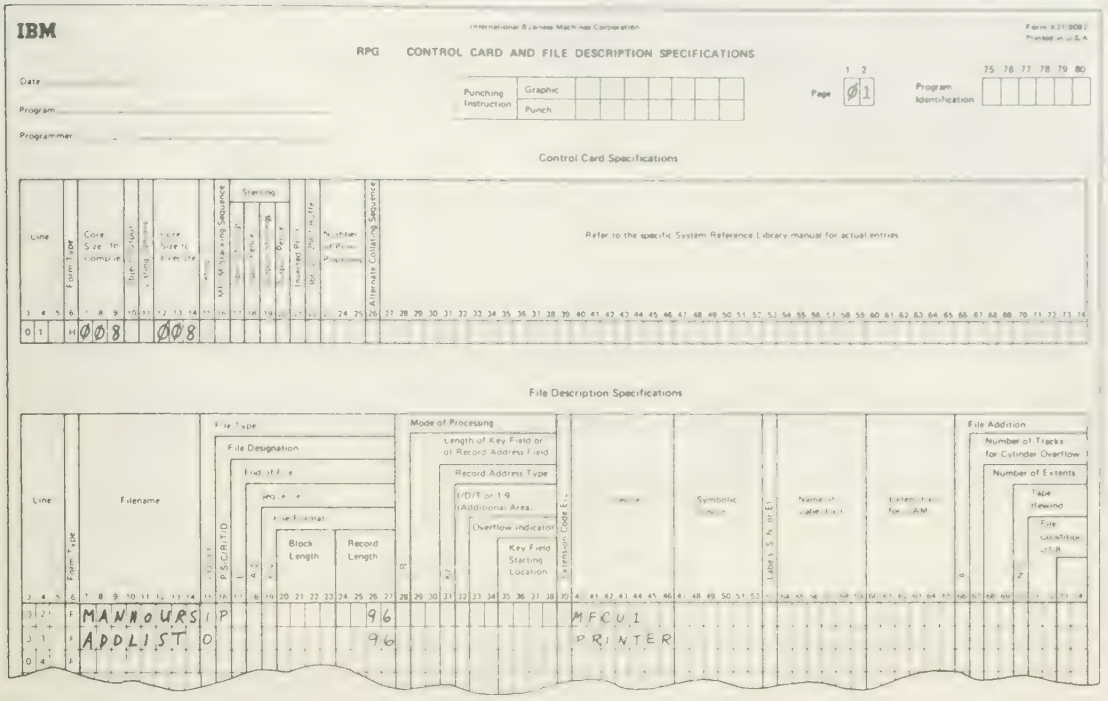
EXAMPLE

Suppose you have a file, called MANHOURS, stored on punched cards. There's one card for each employee:

On each card, columns 1-5 contain the employee's identification number; columns 6-20 contain his name; and columns 21-23 tell how many hours he worked. Let's write an RPG program that makes an IBM System/3 computer print the file on the line printer, with extra spacing, and also print the total number of man-hours in the company, like this:

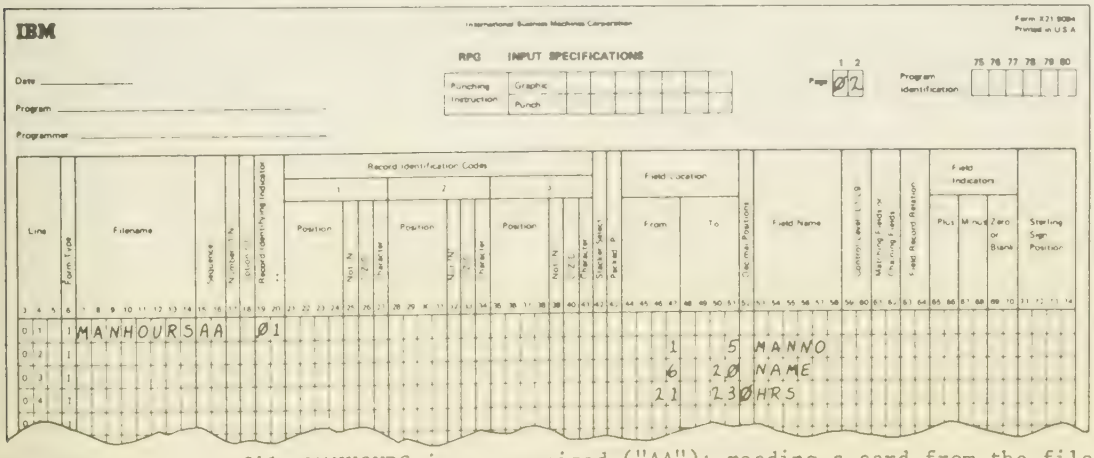


To write the program fill out four forms. The first form describes the controls and files, like COBOL's environment division. Here's the form, filled out correctly:



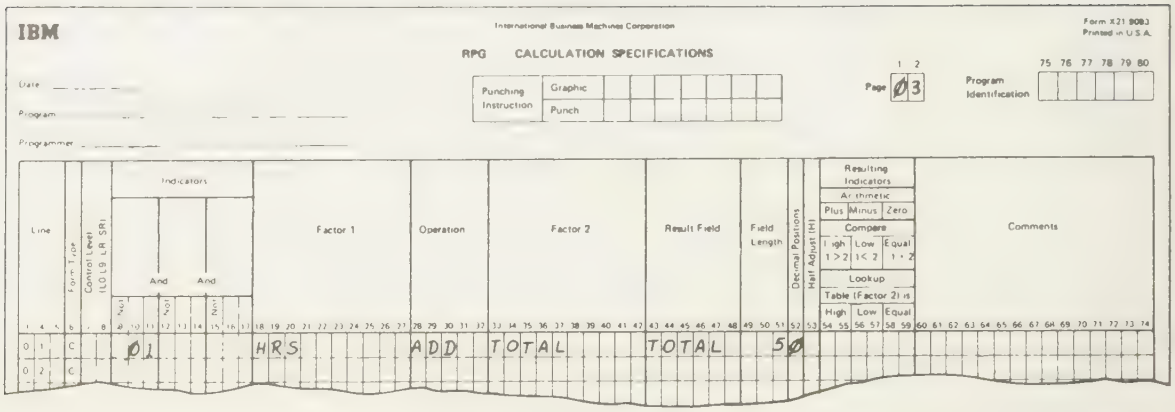
Line 01 says "008 008". That makes the computer reserve 8 kilobytes of memory for the program. Line 02 describes the file MANHOURS; the "IP" means the file is for Input and is the Primary file; the "96" means each card in the file has 96 columns; the "MFCU1" means card reader #1. Line 03 says "ADDLIST" will be the name of the Output file, which has 96 columns and will appear on the PRINTER.

The second form describes the input:



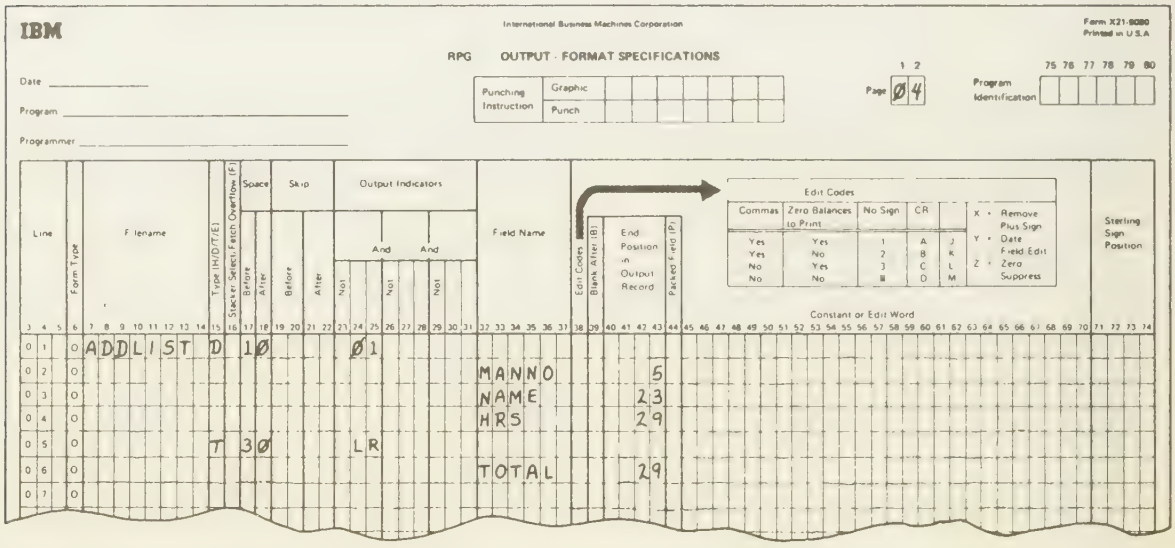
The form says the file MANHOURS is unorganized ("AA"); reading a card from the file is called "activity #01"; on each card, columns 1-5 contain MANNO; columns 6-20 contain NAME, and columns 21-23 contain HRS, a number which has 0 digits after the decimal point.

The third form describes the calculations:



The form says: after each occurrence of activity #01, let HRS + TOTAL be the new TOTAL, which is a 5-digit number having 0 digits after the decimal point.

The fourth form describes the output:



The form explains how to print the file ADDLIST. In line 01, the "D" means "here's how to print each line of Details"; the "10 01" means "press the carriage return 1 time before you print the line, press it 0 times after you print the line, and do the printing after each occurrence of activity #01". Line 02 says to print MANNO so it ends in column 5. Line 03 says to print NAME so it ends in column 23; since the second form said NAME requires 15 columns, the computer will print NAME in columns 9-23. Line 04 says to print HRS so it ends in column 29; since the second form said HRS requires 3 columns, the computer will print HRS in columns 27-29. In line 05, the "T" means "here's how to print the Total"; the "30" means "press the carriage return 3 times before you print the line, and 0 times after"; the "LR" means "print it only after the last card has been read (Last Record)". Line 06 says to print TOTAL so it ends in column 29; since the third form said TOTAL requires 5 columns, the computer will print TOTAL in columns 25-29.

After you've filled out the four forms in longhand, type what you wrote. Here's the RPG program:

```
0101 H008  008
0102 FMANHOURSIP         96            MFCU1                  } from the first form
0103 FADDLIST O          96            PRINTER                }
0201 IMANHOURSAA  01                                          }
0202 I                                          1   5 MANNO   } from the second form
0203 I                                          6  20 NAME    }
0204 I                                         21  230HRS     }
0301 C   01      HRS       ADD  TOTAL     TOTAL   50          } from the third form
0401 OADDLIST D 10     01                                     }
0402 O                          MANNO      5                  }
0403 O                          NAME      23                  } from the fourth form
0404 O                          HRS       29                  }
0405 O        T 30      LR                                    }
0406 O                          TOTAL     29                  }
```

To do that in FORTRAN or COBOL, you'd have to write a loop; you'd have to say GO TO, DO, or PERFORM. RPG makes the computer do loops automatically; you don't have to say so. The order in which you write statements is less important in RPG than in other languages; you're less likely to err; RPG is more reliable.

LISP

TWO VERSIONS

Classical LISP comes in two versions. I'll begin by explaining the most popular, which is called "LISP 1.6" or "EVAL LISP". (For information about the more confusing version, which is called "LISP 1.5" or "EVALQUOTE LISP", see the end of this chapter.)

SIMPLE PROGRAMS

To write a simple LISP program, use just letters, digits, spaces, and parentheses. Do *not* use special symbols such as:

```
+ - * / ↑ "
```

Here's how LISP avoids those symbols:

BASIC	LISP
5+2	(PLUS 5 2)
5-2	(DIFFERENCE 5 2)
5*2	(TIMES 5 2)
5/2	(QUOTIENT 5 2)
5↑2	(EXPT 5 2)
"LOVE"	(QUOTE LOVE)

For example, if you want the computer to add 5 and 2, type this:

```
(PLUS 5 2)
```

When you press the carriage return at the end of that line, the computer will print the answer. (You don't have to say PRINT or RUN or any other special word.) The computer will print:

```
VALUE IS...
7
```

If you type

```
(PLUS 1 2 3 4)
```

the computer will add 1, 2, 3, and 4 and print

```
VALUE IS...
6
```

If you type

```
(DIFFERENCE 7 (TIMES 2 3))
```

the computer will find the difference between 7 and 6 and print

```
VALUE IS...
1
```

If you type —
(QUOTE LOVE)
the computer will print:
VALUE IS...
LOVE

A string must be a single item:

ALLOWED	NOT ALLOWED
(QUOTE LOVE)	(QUOTE LOVE DEATH)

Warning: if you're using Stanford University's LISP 1.6 on a PDP-10 computer, the computer will surprise you. For example, if you think an answer should be thirty-seven the computer will print 45 instead. That's because the computer thinks in base eight: its "45" means "4 eights plus 5", which is thirty-seven.

THE THEORY OF LISTS

LISP can handle lists. Each list must begin and end with a parenthesis. Here's a list of numbers:
(5 7 4 2)
Here's a list of words:
(LOVE HATE WAR PEACE DEATH)

Here's a list of numbers and words:
(2 WOMEN KISS 7 MEN)
That list has five items: 2, WOMEN, KISS, 7, and MEN.

Here's a list of four items:
(HARRY LEMON (TICKLE MY TUBA TOMORROW AT TEN) RUSSIA)
The first item is HARRY; the second is LEMON; the third is a list; the fourth is RUSSIA

In a list, the first item is called the CAR, and the remainder of the list is called the CDR (pronounced "could-er" or "cudder" or "cooder"). For example, the CAR of (LOVE HATE WAR PEACE DEATH) is LOVE, and the CDR is (HATE WAR PEACE DEATH).

CONS is the opposite of CAR and CDR. For example, the CONS of LOVE with (HATE WAR PEACE DEATH) is (LOVE HATE WAR PEACE DEATH).

PROGRAMS THAT MANIPULATE LISTS

To make the computer find the CAR of (SAILORS DRINK WHISKEY), type this:
(CAR (QUOTE (SAILORS DRINK WHISKEY)))
The computer will print:
VALUE IS...
SAILORS

If you type—
(CDR (QUOTE (SAILORS DRINK WHISKEY)))
the computer will print:
VALUE IS...
(DRINK WHISKEY)

If you type—
(CONS (QUOTE SAILORS) (QUOTE (DRINK WHISKEY)))
the computer will find the CONS of SAILORS with (DRINK WHISKEY) and will print:
VALUE IS...
(SAILORS DRINK WHISKEY)

If you type—
(CAR (CDR (QUOTE (SAILORS DRINK WHISKEY))))
the computer will find the CAR of the CDR of (SAILORS DRINK WHISKEY). Since the CDR of (SAILORS DRINK WHISKEY) is (DRINK WHISKEY), whose CAR is DRINK, the computer will print:
VALUE IS...
DRINK

YOUR OWN FUNCTIONS

You can define your own functions. For example, you can define (DOUBLE X) to be 2*X, and (TRIPLE X) to be 3*X, by typing this:

```
(DEFINE (QUOTE (
        (DOUBLE (LAMBDA (X) (TIMES 2 X))
        (TRIPLE (LAMBDA (X) (TIMES 3 X))
)))
```

The first line says you're going to define functions. The next line defines (DOUBLE X) to be (TIMES 2 X). The next line defines (TRIPLE X) to be (TIMES 3 X). The next line closes the parentheses that the top line opened. When you've typed those four lines, the computer will print:
VALUE IS...
(DOUBLE TRIPLE)
Then if you type—
(DOUBLE 4)
the computer will print:
VALUE IS...
8
If you type—
(TRIPLE 5)
the computer will print:
VALUE IS...
15

REPEAT

Let's define REPEAT to be a function, so that (REPEAT (QUOTE LOVE) 5) is (LOVE LOVE LOVE LOVE LOVE), and (REPEAT (QUOTE KISS) 3) is (KISS KISS KISS), and (REPEAT (QUOTE KISS) 0) is ().

If N is 0, (REPEAT X N) is ().

If N is larger than 0, (REPEAT X N) is a list of N X's. That's X followed by N-1 more X's. That's the CONS of X with a list of N-1 more X's. That's the CONS of X with (REPEAT X (DIFFERENCE N 1)). That's (CONS X (REPEAT X (DIFFERENCE N 1))). That's (CONS X (REPEAT X (SUB1 N))), since (SUB1 N) means N-1 in LISP.

So you can define the value of (REPEAT X N) as follows: if N is 0, the value is (); if N is not, the value is (CONS X (REPEAT X (SUB1 N))). Here's what to type:

```
(DEFINE (QUOTE (
        (REPEAT (LAMBDA (X N) (COND
                ((ZEROP N) ())
                (T (CONS X (REPEAT X (SUB1 N))))
        )))
)))
```

The top line says you're going to define a function. The next line says REPEAT is a function of X and N, and the value depends on CONDitions. The next line gives one of the conditions: *if N is ZERO*, the value is (). The next line says: *otherwise*, the value is (CONS X (REPEAT X (SUB1 N))). The next line closes the parentheses opened in the second line. The next line closes the parentheses opened in the top line. The computer will print:

```
VALUE IS...
(REPEAT)
```

Then if you type—

```
(REPEAT (QUOTE LOVE) 5)
```

the computer will print:

```
(LOVE LOVE LOVE LOVE LOVE)
```

The definition is almost circular. In other words, the definition of REPEAT assumes you already know what REPEAT is.

For example, (REPEAT (QUOTE KISS) 3) is defined as the CONS of KISS with (REPEAT (QUOTE KISS) 2). But to understand that definition, you must understand (REPEAT (QUOTE KISS) 2), which is defined as the CONS of KISS with (REPEAT (QUOTE KISS) 1). But to understand *that* definition, you must understand (REPEAT (QUOTE KISS) 1), which is defined as the CONS of KISS with (REPEAT (QUOTE KISS) 0). But to understand *that* definition also, you must understand (REPEAT (QUOTE KISS) 0), which is defined as ().

A definition that is almost circular is called *recursive*. You can say, "The definition of REPEAT is recursive", or "REPEAT is defined recursively", or "REPEAT is defined by recursion", or "REPEAT is defined by induction", or "REPEAT is a recursive function". You can write recursive definitions in LISP, ALGOL, and several other languages (such as PL/I and SNOBOL), but not in BASIC or FORTRAN.

When the computer uses a recursive definition, the computer refers to the definition *repeatedly* before getting out of the circle. Since the computer repeats, it is performing a loop. In BASIC and FORTRAN, the only way to make the computer perform a loop is to say GO TO or FOR or DO. Although LISP contains a go-to command, LISP programmers avoid it, and write recursive definitions instead.

ITEM

As another example of recursion, let's define the function ITEM so that (ITEM N X) is the Nth item in list X. Thus (ITEM 4 (QUOTE (LOVE HATE WAR PEACE DEATH))) should be the 4th item of (LOVE WAR PEACE DEATH), which is PEACE.

If N is 1, (ITEM N X) is the first item in X, which is the CAR of X, which is (CAR X).

If N is larger than 1, (ITEM N X) is the Nth item in X. That's the (N-1)th item in the CDR of X. That's (ITEM (SUB1 N) (CDR X)).

So define the value of (ITEM N X) as follows: if N is 1, the value is (CAR X); if N is not, the value is (ITEM (SUB1 N) (CDR X)). Here's what to type:

```
(DEFINE (QUOTE (
        (ITEM (LAMBDA (N X) (COND
                ((ONEP N) (CAR X))
                (T (ITEM (SUB1 N) (CDR X)))
        )))
)))
```

If your computer doesn't understand (ONEP N), say (EQUAL 1 N) instead.

WHY USE LISP?

LISP has three advantages over FORTRAN and BASIC:

1. LISP allows recursive definitions. FORTRAN and BASIC do not.
2. LISP lets you handle lists flexibly: you can make a list so some of its items are strings, some items are numbers, and some items are themselves lists of varying lengths—short lists, long lists, and lists of lists (of lists of lists of lists...). Your program can change the list's structure: you can make the list longer, then shorter, then replace one of its items by a list of other items, etc. FORTRAN and BASIC permit only list structures that are simple, described in the DIMENSION statement and never changed.
3. LISP makes the computer periodically delete from its memory all the variables you are no longer using, and use that memory to hold new variables instead. That activity is called *garbage collection*. LISP makes the computer do it automatically, without the programmer having to say so. To collect garbage in BASIC and FORTRAN, you'd have to explain to the computer exactly how and when, by writing a garbage-collecting subroutine, which is difficult.

For those reasons, LISP is the most popular language among researchers in artificial intelligence, who try to make the computer win checkers and chess, prove theorems, understand English, and control a robot's muscles, sense organs, and thinking.

IMPROVEMENTS ON LISP

There are many variations of LISP. QLISP, PLANNER, and CONNIVER contain extra features that make the computer prove theorems. For example, suppose you want to solve this problem, by using PLANNER:

```
Nixon is human.
Socrates is human.
Socrates is Greek.
All humans are fallible.
Is there a fallible Greek?
```

Type this:

```
(HUMAN NIXON)
(HUMAN SOCRATES)
(GREEK SOCRATES)
(THCONSE (X) (FALLIBLE $?X) (THGOAL (HUMAN $?X)))
(THPROG (X)
        (THGOAL (FALLIBLE $?X) (THTBF THTRUE))
        (THGOAL (GREEK $?X))
```

The computer will print SOCRATES.

LISP requires ridiculously many parentheses and uses awkward symbolism, such as QUOTE, LAMBDA, and COND. To remedy those flaws, several improvements have been invented: MLISP2 (which will be expanded to form LISP70) and TREET combine LISP's power with ALGOL's simplicity. LOGO combines elements from LISP and BASIC, and adds extra features so you can produce art and music and control robots that look like turtles.

LISP 1.5

I explained how to write commands in LISP 1.6 (or EVAL LISP). To write a command in LISP 1.5 (or EVALQUOTE LISP), make two changes:

1. Begin the command with a word, instead of a parenthesis. For example, instead of saying (PLUS 2 2), say PLUS (2 2). Instead of saying (DIFFERENCE 7 (TIMES 2 3)), say DIFFERENCE (7 (TIMES 2 3)).
2. Omit the word QUOTE and its parentheses, unless the QUOTE is in a command that's in a command. For example, instead of CAR ((QUOTE (SAILORS DRINK WHISKEY))), say just CAR ((SAILORS DRINK WHISKEY)). But in CAR ((CDR (QUOTE (SAILORS DRINK WHISKEY)))), you must keep the QUOTE, because it is in the command CDR which is in the command CAR.

Write a function definition in LISP 1.5 the same as in LISP 1.6, except for the top and bottom lines:

TOP AND BOTTOM LINES IN LISP 1.6

```
(DEFINE (QUOTE (
)))
```

TOP AND BOTTOM LINES IN LISP 1.5

```
DEFINE ((
))
```

Leave the middle lines unchanged.

COBOL

HOW COBOL DEVELOPED

From 1954 to 1958, Sperry Rand Corp. (which makes UNIVAC computers) developed a language called FLOW-MATIC, to solve problems in business. In 1958, the Air Force developed a variation called AIMACO. From 1957 to 1959, IBM developed a competing language, called COMMERCIAL TRANSLATOR. From 1959 to 1960, Honeywell Corp. developed another competing language, called FACT.

In April 1959, a group of programmers and manufacturers met at the University of Pennsylvania and decided it would be better to have a *single* business-oriented language for *all* computers. The group arranged for a larger group to meet at the Pentagon in May. At the May meeting, the large group created a committee that later created a subcommittee that created COBOL.

From FLOW-MATIC, COBOL borrowed the idea of beginning each statement with an English verb, and separating the data descriptions from the procedures. From COMMERCIAL TRANSLATOR, COBOL borrowed group items (01 and 02), PICTURE symbols, fancy IF THEN, and COMPUTE formulas.

The members of the COBOL committee argued with each other. Some members wanted COBOL to let the programmer construct mathematical formulas by using these symbols:
+ - * / = ( )
But other members of the committee disagreed: they said that since COBOL is supposed to be for stupid businessmen who fear formulas, COBOL ought to use the words ADD, SUBTRACT, MULTIPLY, and DIVIDE instead. The committee compromised: when you write a COBOL program, you can use the words ADD, SUBTRACT, MULTIPLY, and DIVIDE, or a formula; but if you use a formula, you must warn the computer, by putting the word COMPUTE in front of the formula.

COBOL can handle short numbers. Can it handle long numbers also? How long? How many digits? The committee decided that COBOL would handle any number up to 18 digits long. The committee also decided that COBOL would handle any variable name up to 30 characters long. So the limits of COBOL are "18 and 30". Why did the committee pick those two numbers—"18 and 30"—instead of "16 and 32"? Answer: some manufacturers wanted "16 and 32" (because their computers were based on the numbers 16 and 32), but other manufacturers wanted other combinations (such as "24 and 36"); the committee, hunting for a compromise, chose "18 and 30", because *nobody* wanted it, and it would therefore give no manufacturer an unfair advantage over competitors. In other words, COBOL was designed to be equally terrible for everybody! That's politics!

In 1960, the Department of Defense announced it would buy only computers that understand COBOL, unless a manufacturer can demonstrate why COBOL would not be helpful. In 1961, Westinghouse Electric Corp. made a similar announcement. Other companies followed. COBOL became the most popular computer language.

COBOL 60 was improved, to form COBOL 61. A "Report Writer" feature and the verb SORT were introduced in 1962. Then came COBOL 65, COBOL 68, and COBOL 74.

Most large computers still use COBOL 68. To write business programs on a minicomputer, you have to use a *small* part of COBOL 74, or BASIC, or FORTRAN, or RPG (on IBM computers), or DIBOL (on PDP computers), or DATABUS (on Datapoint computers). No computer understands *full* COBOL 74 yet.

COBOL'S FLAWS

CONOL has some little flaws. For example, according to COBOL, the data division's FD must contain information about labeling. But labeling describes the *environment*, not the *data*, and therefore ought to belong in the environment division instead. Jean Sammet, who headed the subcommittee that invented COBOL, admits her subcommittee made a mistake about where to put the labeling. But it's too late to change COBOL no

COBOL'S POPULARITY

COBOL remains the most popular language for large computers. IBM tried to make programmers switch from COBOL to PL/I, but didn't succeed.

For minicomputers and microcomputers, programmers have switched to BASIC.

GPSS

WHAT DOES GPSS ANALYZE?

A *queue* is a line of people who are waiting. GPSS analyzes queues.

EXAMPLE

For example, let's use GPSS to analyze the customers in "Quickie Joe's Barbershop". Joe is the only barber in the shop, and he spends exactly 7 minutes on each haircut. (That's why he's called "Quickie Joe".) About once every 10 minutes, a new customer enters the barbershop; more precisely, the number of minutes before another customer enters is a random number between 5 and 15. Let's make the computer imitate the barbershop, and summarize what happens to the first 100 customers.

Here's the program:

```
      SIMULATE
      GENERATE    10,5    A new customer comes every 10 minutes ± 5 minutes.
      QUEUE       JOEQ    He waits in the queue, called JOEQ.
      SEIZE       JOE     When his turn comes, he seizes JOE,
      DEPART      JOEQ      which means he leaves the JOEQ.
      ADVANCE     7       After 7 minutes go by,
      RELEASE     JOE       he releases JOE (so someone else can use JOE)
      TERMINATE   1         and leaves the shop.
      START       100     Do all that 100 times.
      END
       ↑           ↑
     8th         19th
   column      column
```

When you run the program, the computer will print many numbers. For example, it will tell you that Joe was working 68.5% of the time (the rest of the time, his shop was empty and he was waiting for customers). There was never more than 1 customer waiting. "On the average", .04 customers were waiting. There were 101 customers (the 101st customer stops the experiment); 79 of them (78.2% of them) obtained Joe immediately and didn't have to wait. The "average customer" had to wait in line .405 minutes. The "average not-immediately-served customer" had to wait in line 1.863 minutes.

Below the RELEASE statement and above the TERMINATE statement, you can insert two extra statements:

```
        TABULATE    1
1       TABLE       M1,0,1,26
```

↑
2nd column

They make the computer print even more information. The computer will say that of the 100 serious customers, the "average customer" spent 7.369 minutes in the shop (from when he walked in to when he walked out). More precisely, 79 customers spent 7 minutes each; 9 customers spent 8 minutes each; 9 customers spent 9 minutes each; 2 customers spent 10 minutes each; and 1 customer had to spend 11 minutes. The computer also prints the "standard deviation", "cumulative tables", and other statistical claptrap.

On your own computer, the numbers might be slightly different, depending on how the random numbers came out. To have more faith in the computer's averages, try 1000 customers instead of 100.

ALTERNATIVE LANGUAGES

For most problems about queues, GPSS is the easiest language to use. But if your problem is *very* complicated, you might have to use SIMSCRIPT (based on FORTRAN) or SIMULA (an elaboration of ALGOL) or SIMPL/I (an elaboration of PL/I).

SIMPLE EXAMPLES

Here's a SNOBOL program:

```
        A = -2
        B = A + 10.6
        C = "BODY TEMPERATURE IS 9" B
        OUTPUT = "MY " C
END
```

Indent each line except END. Indent *at least* one space; you can indent more spaces if you wish. Put spaces around the symbol =, the symbol +, and other operations. The first line says A is the integer -2. The next line says B is the real number 8.6 The next line says C is the string "BODY TEMPERATURE IS 98.6". The next line makes the computer print:

MY BODY TEMPERATURE IS 98.6

In SNOBOL, a variable's name can be short (like A or B or C) or very long (as long as you wish). For long names, SNOBOL uses periods instead of hyphens:

COBOL	SNOBOL
NUMBER-OF-BULLIES-I-SQUIRTED	NUMBER.OF.BULLIES.I.SQUIRTED

This program's more advanced:

```
LOOP    OUTPUT = "CAT"
        OUTPUT = "DOG" :(LOOP)
END
```

The first line (whose name is LOOP) makes the computer print:

CAT

The next line makes the computer print—

DOG

and then go to LOOP. Altogether the computer will print:

CAT
DOG
CAT
DOG
CAT
DOG
etc.

REPLACEMENT

SNOBOL lets you easily replace one phrase by another.

```
        X = "SIN ON A PIN WITH A DIN"
        X "IN" = "UCK"
        OUTPUT = X
END
```

The first line says X is the string "SIN ON A PIN WITH A DIN". The next line says: in X, replace the first "IN" by "UCK". So X becomes "SUCK ON A PIN WITH A DIN". The next line says the output is X, so the computer will print:

```
SUCK ON A PIN WITH A DIN
```

That program changed the *first* "IN" to "UCK". Here's how to change *every* "IN" to "UCK":

```
        X = "SIN ON A PIN WITH A DIN"
        X "IN" = "UCK"
        X "IN" = "UCK"
        X "IN" = "UCK"
        OUTPUT = X
END
```

The first line says X is "SIN ON A PIN WITH A DIN". The second line replaces an "IN" by "UCK", so X becomes "SUCK ON A PIN WITH A DIN". The next line replaces another "IN" by "UCK", so X becomes "SUCK ON A PUCK WITH A DIN". The next line replaces another "IN", so X becomes "SUCK ON A PUCK WITH A DUCK", which the next line prints.

This program does the same thing:

```
        X = "SIN ON A PIN WITH A DIN"
LOOP    X "IN" = "UCK" :S(LOOP)
        OUTPUT = X
END
```

The first line says X is "SIN ON A PIN WITH A DIN". The next line replaces "IN" successfully, so X becomes "SUCK ON A PIN WITH A DIN". At the end of the line, the `:S(LOOP)` means: if Successful, go to LOOP. So the computer goes back to LOOP. The computer replaces "IN" successfully again, so X becomes "SUCK ON A PUCK WITH A DIN", and the computer goes back to LOOP. The computer replaces "IN" successfully again, so X becomes "SUCK ON A PUCK WITH A DUCK", and the computer goes back to LOOP. The computer *tries* to replace "IN" again; but since X doesn't contain an "IN" now, the computer does not succeed. So the computer ignores the `:S(LOOP)`, and proceeds instead to the next line, which prints:

```
SUCK ON A PUCK WITH A DUCK
```

DELETING

This program deletes the first "IN":

```
        X = "SIN ON A PIN WITH A DIN"
        X "IN" =
        OUTPUT = X
END
```

The second line says to replace an "IN" by nothing. In other words, it deletes an "IN". X becomes "S ON A PIN WITH A DIN", which the computer will print.

This program deletes *every* "IN":

```
        X = "SIN ON A PIN WITH A DIN"
LOOP    X "IN" = :S(LOOP)
        OUTPUT = X
END
```

The computer will print:

```
S ON A P WITH A D
```

COUNTING

Let's count how often "IN" appears in "SIN ON A PIN WITH A DIN". Here's how.... Delete each "IN"; but each time you delete one, increase the COUNT by 1. Here's the program:

```
        X = "SIN ON A PIN WITH A DIN"  Define X.
        COUNT = 0                      Start the COUNT at 0.
LOOP    X "IN" = :F(ENDING)            Try to delete an "IN"; if Failing, go to ENDING.
        COUNT = COUNT + 1 :(LOOP)      Increase the COUNT by 1, and go back to LOOP.
ENDING  OUTPUT = COUNT                 Print the COUNT.
END
```

The third line tries to delete an "IN": *if successful*, the computer proceeds to the next line, which increases the COUNT and goes back to LOOP; but *if failing* (because no "IN" remains), the computer goes to ENDING, which prints the COUNT. The computer will print:

```
3
```

HOW SNOBOL DEVELOPED

At MIT during the 1950's, Noam Chomsky invented a notation called *transformational-generative grammar*, which helps linguists analyze English and translate betweeen English and other languages. His notation is as important to linguists as algebra is to scientists. (A decade later, he became even more famous, for starting the rebellion against the Vietnam War.)

In 1957 and 1958, his colleague Victor Yngve developed a computer language called COMIT, which uses Chomsky's notation. COMIT is as convenient for linguists as FORTRAN is for scientists.

At Bell Telephone Laboratories in 1962, Chester Lee invented a similar language, called Symbolic Communication Language (SCL). Like COMIT, it manipulated string. COMIT manipulated strings of *words*, but SCL manipulated strings of *mathematical symbols*. SCL was used for abstract mathematics.

A team at Bell Telephone Laboratories decided to invent a language, similar to SCL, but easier to learn and including features from COMIT. What should they name their new language? At first, they called it SCL7, because it resembled SCL. Then they changed the name to SEXI (which stands for String EXpression Interpreter), but the management of Bell Telephone Laboratories didn't like sex. Then, as a joke, they named it SNOBOL, using the flimsy excuse that SNOBOL stands for StriNg-Oriented symBOlic Language.

Cynics jeered that SNOBOL didn't have "a snowball's chance in Hell". But the cynics were wrong, and SNOBOL became popular. It was used mainly for writing programs that translate between computer languages. (For example, you could write a SNOBOL program that translates FORTRAN into BASIC.)

Which is better: COMIT or SNOBOL? People who like Chomsky's notation (such as linguists) prefer COMIT. People who like algebra (such as scientists) prefer SNOBOL.

SNOBOL's supporters were more active than COMIT's: they produced SNOBOL 2, SNOBOL 3, SNOBOL 4, and SNOBOL 4B, taught SNOBOL to the newest computers, wrote many books about SNOBOL, and emphasized that SNOBOL can solve *any* problem about strings, even if the problem has nothing to do with linguistics. They won: most people use SNOBOL instead of COMIT, though COMIT might still make a comeback.

Today, most versions of SNOBOL are named after baseball pitching methods—such as FASBOL, SLOBOL, and SPITBOL. (SPITBOL stands for SPeedy ImplemenTation of snoBOL.)

BASIC

WHO IS KEMENY?

BASIC's success is due largely to John Kemeny, a colorful character who has changed many lives, including mine. Here is the saga of Kemeny....

In 1926, he was born as a Hungarian Jew. In 1940, he and his parents fled to America, to escape the Nazis. When he began high school in New York, he knew hardly any English; nevertheless, he graduated as the top student in the class. Four years later, he graduated from Princeton *summa cum laude*, even though 1½ of those years he had to spend in the Army, where he helped solve equations for the atomic bomb.

Two years after his B.A., he got a Ph.D. in mathematics *and* philosophy, because his thesis on symbolic logic combined both fields. While working for the Ph.D., he was also Einstein's youngest assistant. But he told Einstein he wanted to drop mathematics, and instead hand out leaflets for World Peace. Eistein said: handing out leaflets would waste his talents; the best way for him to work for World Peace would be to become a famous mathematician, so people would *listen* to him, as they had to Einstein. He took Einstein's advice, and stayed with math.

After getting his Ph.D., he taught symbolic logic in Princeton's philosophy department, and had the good fortune to bump into J. Laurie Snell, who taught statistics in the math department. Here's how they met....A student walked up to Snell after class and asked, "Is it true that if I stand at Times Square long enough, I'll meet my long-lost brother?" Snell asked, "What are you talking about? Who said so?" The student said, "Professor Kemeny in the philosophy department". Kemeny had been giving a philosophical lecture on "random-walk" probabilities.

In 1953, most of Dartmouth College's math professors were retiring, so Dartmouth asked Kemeny to come to Dartmouth, chair the department, and "bring all your friends". He accepted the offer and brought his friends. That's how Dartmouth stole Princeton's math department.

He discovered an easier method for handling "Markov chain" probabilities. He told Snell (whom he had stolen). Snell asked, "Can your method prove *this* theorem...?" Kemeny proved it. Snell asked, "Can your method prove *this* theorem also...?" Kemeny proved it. By that kind of dialogue, Kemeny and Snell co-authored *Finite Markov Chains*, which in turn led to *Finite Mathematics* and *Finite Mathematical Structures*, co-authored by Kemeny, Mirkil, Snell, and Thompson. In their honor, the department's vice-chairman named his pig "Sir Kemeny-Mirkil-Snell-Thompson". Mirkil committed suicide, though not because of the pig.

Kemeny's department got General Electric to sell Dartmouth a computer at a 90% discount, in return for which Dartmouth had to invent programs for it and let General Electric use them. To write the programs, Kemeny invented his own little computer language in 1963 and showed it to his colleague Thomas Kurtz, who knew less about philosophy but more about computers. Kurtz added features from ALGOL and FORTRAN and called the combination "BASIC".

After inventing BASIC, Kemeny became bored, and thought of quitting Dartmouth. Then Dartmouth asked him to become president of the college. He accepted.

Later, when the Three-Mile Island nuclear power plant almost exploded, President Jimmy Carter told Kemeny to head the investigation, because of Kemeny's reputation for profound philosophical and scientific impartiality. Kemeny's report was impartial—and sharply critical of the nuclear industry.

WHY BASIC?

BASIC is simpler than both ALGOL and FORTRAN in two ways:

1. In ALGOL and FORTRAN, you must tell the computer which variables are integers and which are reals. In ALGOL, you do that by saying INTEGER or REAL. In FORTRAN you do that by choosing an appropriate first letter for the variable's name. In BASIC, the computer assumes all variables are real.
2. In ALGOL and FORTRAN, output is a hassle. In FORTRAN, you have to worry about FORMATs. In ALGOL, each computer handles output differently—and in most cases strangely. BASIC's PRINT statement automatically invents a good format.

Is BASIC closer to ALGOL than to FORTRAN? On the one hand, BASIC uses the ALGOL words FOR, STEP, and THEN and the ALGOL symbol ↑. On the other hand, BASIC uses the FORTRAN words RETURN and DIMENSION (abbreviated DIM); and BASIC's "FOR I = 1 TO 9 STEP 2" puts the step size at the *end* of the statement, like FORTRAN's "DO 30 I = 1,9,2" and unlike ALGOL's "FOR I:=1 STEP 2 UNTIL 9".

BASIC is *not* the simplest computer language. "JOSS", which was developed a year earlier by the RAND Corporation, is simpler to learn. But JOSS doesn't have string variables and doesn't name programs (you must give each program a number instead, and remember what the number was). Also, programs written in JOSS run more slowly and require more of the computer's memory than if written in BASIC.

SIX VERSIONS

Kemeny and Kurtz finished the original version of BASIC in May 1964. It included only these statements: IF...THEN, FOR...NEXT, DATA...READ, GO SUB...RETURN, GO TO, PRINT, LET, REM, DIM, DEF, and END.

In that version, the only punctuation allowed in the PRINT statement was the comma. The second version of BASIC (October 1964) added the semicolon.

The third version (1966) added the words INPUT, RESTORE, and MAT. Kemeny included fancy MAT functions (such as TRN, IDN, and INV), to help his students compute Markov-chain MATrix probabilities.

In all those versions, you could use variables. For example, you could say LET X=3. A variable was a letter that stood for a number. The fourth version (1967) added a new concept: string variables (such as A$). That version also added TAB (to improve the printing), RANDOMIZE (to improve RND), and ON...GO TO.

The fifth version (1970) added data files (sequential access and random access).

The sixth version (1971) added PRINT USING and a sophisticated way to handle subroutines—a way so sophisticated that most microcomputers don't have it yet!

HOW BASIC BECAME POPULAR

Kemeny and Kurtz worked on BASIC with a fervor that was almost religious. They believed *every* college graduate should know how to program a computer, and be as literate in BASIC as in English. They convinced Dartmouth to spend as much on its computer as on the college library. They put computer terminals in practically every college building (even in the dorms), and let all the kids who lived in the town come onto the campus and join the fun. Altogether, the campus has about 300 terminals. 90% of all Dartmouth students use BASIC before they graduate.

Dartmouth trained high-school teachers how to use BASIC. Soon many colleges, high schools, and prep schools throughout New England had terminals connected to Dartmouth's computer via telephone.

When Digital Equipment Corp. sold PDP-8 computers that understood both BASIC and FOCAL (a stripped-down version of JOSS), programmers said they preferred the BASIC. So Digital chose BASIC to be the main language for its PDP-11, which has become the most popular minicomputer in the world.

Inspired by the PDP-8 and PDP-11 minicomputers, Bill Gates wrote a version of BASIC for *micro*computers; he called it Microsoft BASIC. It became an instant success.

WILL BASIC REPLACE FORTRAN?

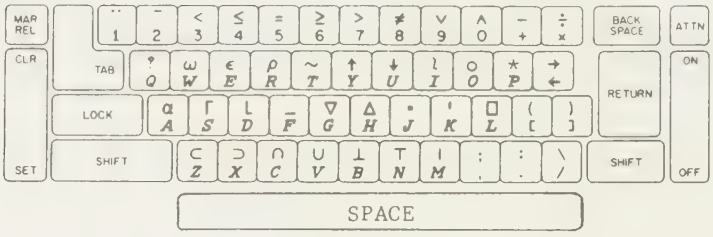
Since BASIC is simpler than FORTRAN, many FORTRAN programmers are switching to BASIC. But some scientists still use FORTRAN, for these reasons:

1. FORTRAN allows long variable names (6 letters). Some versions of BASIC do not.
2. A program will run more quickly in FORTRAN than in BASIC, if the program contains many loops or will be used by many people.
3. FORTRAN allows complex numbers.
4. FORTRAN lets you choose your own format for input.
5. In FORTRAN's CALL statement, you can use parentheses. In BASIC's GOSUB statement, parentheses are forbidden.
6. All versions of FORTRAN are decent. Some versions of BASIC are lousy—especially the version on IBM maxicomputers. (IBM spurns BASIC almost as much as ALGOL, and for the same reason: the language was invented by non-IBMers.)
7. By the time BASIC was invented, many FORTRAN programs had been written already. To improve those programs, it's easier to add a few more lines of FORTRAN than to translate the whole program into BASIC.

In spite of those seven arguments for FORTRAN, BASIC's popularity is increasing rapidly. In the future, more scientists will use BASIC than FORTRAN, though the toughest programs will still require FORTRAN.

SIMPLE CALCULATIONS

To use APL easily, type on a special terminal, whose keyboard looks like this:



To compute 8+9, type this:

```
      8+9
```

When you press the carriage return at the end of that line, the computer will print the answer. (You don't have to say RUN or PRINT or any other special word.) The computer will print:

```
17
```

Altogether, the conversation looks like this:

```
      8+9
17
```

What you type (the 8+9) is indented. The computer's answer (the 17) begins at the left margin. Whenever it's your turn to type, the computer automatically presses the space bar six times, so what you type will be indented six spaces.

SCALAR OPERATORS

APL uses these *scalar operators*:

APL NAME	SYMBOL	MEANING
plus	A+B	add
minus	A−B	subtract
negative	−B	negative
times	A×B	multiply
signum	×B	1 if B>0; ¯1 if B<0; 0 if B=0
divide	A÷B	divide
reciprocal	÷B	$\frac{1}{B}$
power	A*B	A^B; in other words, A raised to the Bth power
exponential	*B	e^B, where e is 2.718281828459045
log	A⍟B	$\log_A B$
natural log	⍟B	$\log_e B$
maximum	A⌈B	A or B, whichever is larger
ceiling	⌈B	B rounded up to an integer
minimum	A⌊B	A or B, whichever is smaller
floor	⌊B	B rounded down to an integer
residue	A|B	the remainder when you divide A into B; for example, 4|19 is 3
magnitude	|B	the absolute value of B
binomial	A!B	how many A-element subsets you can form from a set of B elements; for example, 3!5 is 10
factorial	!B	1 times 2 times 3 times 4 times ... times B
roll	?B	a random integer from 1 to B
circular	A○B	$\sqrt{1-B^2}$ if A=0; sin B if A=1; cos B if A=2; tan B if A=3; $\sqrt{1+B^2}$ if A=4; sinh B if A=5; cosh B if A=6; tanh B if A=7; arcsin B if A=¯1; arccos B if A=¯2; arctan B if A=¯3; $\sqrt{B^2-1}$ if A=¯4; arcsinh B if A=¯5; arccosh B if A=¯6; arctanh B if A=¯7
pi times	○B	π times B
less	A<B	1 if A is less than B; otherwise 0
greater	A>B	1 if A is greater than B; otherwise 0
equal	A=B	1 if A equals B; otherwise 0
not less	A≥B	1 if A is greater than or equal to B; otherwise 0
not greater	A≤B	1 if A is less than or equal to B; otherwise 0
not equal	A≠B	1 if A is not equal to B; otherwise 0
and	A∧B	1 if A and B are both 1; otherwise 0
or	A∨B	1 if A or B is 1; otherwise 0
not	~B	1 if B is 0; otherwise 0
nand	A⍲B	1 if A and B are not both 1; otherwise 0
nor	A⍱B	1 if neither A nor B is 1; otherwise 0

To make the symbol ⍟, type the symbol *, then press the BACK SPACE key, then type the symbol ○.

ORDER OF OPERATIONS

The computer does all calculations *from right to left*. For example, if you type—

```
2×3+5
```

the computer will start with 5, add 3 (to get 8), and then multiply by 2 (to get 16). The computer will print:

16

In BASIC, FORTRAN, and most other languages, the answer would be 11 instead.

If you type—

```
9-4-3
```

the computer will start with 3, subtract it from 4 (to get 1), and then subtract from 9 (to get 8). The computer will print:

8

In most other computer languages, the answer would be 2 instead.

You can use parentheses. Although 9-4-3 is 8, (9-4)-3 is 2.

Compare these examples:

-4+6 is ¯10

¯4+6 is 2

In both examples, the 4 is preceded by a negative sign; but in the second example, the negative sign is raised, so it is as high as the 4. (To make the raised negative, hold down the SHIFT key while you type a 2. To make the regular negative, hold down the SHIFT key while you type a +.) The first example makes the computer start with 6, add 4 (to get 10), and then negate it (to get ¯10). The second example makes the computer start with 6, and add a ¯4; the answer is 2.

DOUBLE PRECISION

APL is super-accurate. It does all calculations by using double-precision.

VARIABLES

You can use variables:

```
X←3
X+2
```

The first line says X is 3. The second line makes the computer print X+2. The compute will print:

5

A variable's name can be long—up to 77 letters and digits. The name must begin with a letter.

VECTORS

A variable can stand for a list of numbers:

```
Y←5 2 8
Y+1
```

The first line says Y is the *vector* 5 2 8. The next line makes the computer add 1 to each item and print:

6 3 9

This program prints the same answer:

```
5 2 8+1
```

The computer will print:

6 3 9

This program prints the same answer:

```
      1+5 2 8
```

You can add a vector to another vector:

```
      A←5 2.1 6
      B←3 2.8 ¯7
      A+B
```

The computer will add 5 to 3, and 2.1 to 2.8, and 6 to ¯7, and print:

```
8  4.9  ¯1
```

This program prints the same answer:

```
      5 2.1 6+3 2.8 ¯7
```

This program prints the same answer:

```
      A←5 2.1 6
      B←3 2.8 ¯7
      C←A+B
      C
```

Here's something a bit different:

```
      X←4 2 3
      +/X
```

The first line says X is the vector 4 2 3. The next line makes the computer print the sum, 9.

This program prints the same answer:

```
      Y←+/4 2 3
      Y
```

You can combine many ideas on the same line, but remember that the computer goes from right to left:

```
      219-1 4 3+6×+/5 1 3×2 4 7
```

The computer will start with 2 4 7, multiply it by 5 1 3 (to get 10 4 21), find the sum (which is 35), multiply by 6 (to get 210), add 1 4 3 (to get 211 214 213), and then subtract from 219 (to get 8 5 6). The computer will print:

```
8  5  6
```

Each of APL's scalar operators works similarly to addition. Here are a few examples:

2 4 10×3 7 9 is 6 28 90
÷2 4 10 is .5 .25 .1
-2 4 10 is ¯2 ¯4 ¯10
×/2 4 10 is 2×4×10, which is 80
-/9 5 3 is 9-5-3, which is 7 (since the computer works from right to left)
⌊/6.1 2.7 4.9 is 6.1⌊2.7⌊4.9, which is 2.7 (since ⌊ means minimum)
⌊6.1 2.7 4.9 is ⌊6.1 followed by ⌊2.7 followed by ⌊4.9, which is 6 2 4 (since ⌊ means floor)

VECTOR OPERATORS

Here are *vector operators*; the examples assume V is 8 5 2 7:

APL NAME	SYMBOL	VALUE	REASON
size	⍴V	4	V has 4 items
reshape	9⍴V	8 5 2 7 8 5 2 7 8	make 9 items from V
catenate	V,6 4	8 5 2 7 6 4	V followed by 6 4
index	V[3 1]	2 8	V's 3rd item and 1st item
reverse	⌽V	7 2 5 8	reverse V
rotate	1⌽V	5 2 7 8	rotate V, by beginning after the 1st item
index generator	⍳4	1 2 3 4	count up to 4
index of	V⍳2 8 9	3 1 5	in V, find 2, 8, and 9: 2 is V's 3rd item; 8 is V's 1st item; 9 isn't in V (and is therefore a "5th" item)
take	3↑V	8 5 2	the first 3 items from V
take	¯3↑V	5 2 7	the last 3 items from V
drop	3↓V	7	omit the first 3 items from V
drop	¯3↓V	8	omit the last 3 items from V
grade up	⍋V	3 2 4 1	V's smallest item is 2, V's 3rd item; the next smallest is 5, V's 2nd item; the next smallest is 7, V's 4th item; the largest is 8, V's 1st item
grade down	⍒V	1 4 2 3	V's largest item is 8, V's 1st item; the next largest is 7, V's 4th item; the next largest is 5, V's 2nd item; the smallest is 2, V's 3rd item
membership	7 5 6∈V	1 1 0	find whether 7, 5, and 6 are in V; the answers are: yes, yes, no
compress	1 0 0 1/V	8 7	take part of V, using this pattern: take, omit, omit, take
expand	1 0 0 1 1 1\V	8 0 0 5 2 7	insert zeros into V, using this pattern: item, 0, 0, item, item, item
decode	10⊥V	8527	(((8)10+5)10+2)10+7
decode	0 7 24 60⊥V	87967	(((8)7+5)24+2)60+7; in other words, if a week is 7 days, and a day is 24 hours, and an hour is 60 minutes, then 8 weeks 5 days 2 hours 7 minutes is 87967 minutes
encode	0 7 24 60⊤87967	8 5 2 7	87967 minutes is 8 weeks 5 days 2 hours 7 minutes
deal	3?10	depends	a list of 3 different random integers (no repetitions); the integers are small (from 1 to 10)

LOVE OR HATE?

Some programmers *love* APL, because its notation is brief. Others hate APL, because its notation is hard for a human to read. The haters seem to be winning, and the percentage of programmers who use APL is decreasing.

HOW PL/I DEVELOPED

In 1963, IBM began inventing a new language that would improve on traditional FORTRAN. At first, IBM called the new language "FORTRAN VI". As work on it progressed, IBM realized it would be so different from traditional FORTRAN that it should have a different name. In 1964, IBM changed the name to "NPL" (New Programming Language). When IBM discovered that the letters "NPL" already stood for the National Physics Laboratory in England, IBM changed the name to "PL/I" (Programming Language One).

STATEMENTS

PL/I uses these statements for input and output:

STATEMENT'S FIRST WORD	WHAT THE COMPUTER WILL DO
GET	input from a terminal or from a typical file
PUT	print on a terminal or on a typical file
OPEN	start using a file
CLOSE	stop using a file
READ	input from a file whose picture is unedited
WRITE	print on a file whose picture is unedited
DELETE	delete an item from a file
REWRITE	replace an item in a file
FORMAT	use a certain form for spacing the input and output
DISPLAY	print on the computer center's main terminal (used by the operator), and wait for the operator to reply
LOCATE	print a "based" variable onto a file
UNLOCK	let other programs use the file

These statements interrupt:

STATEMENT'S FIRST WORD	WHAT THE COMPUTER WILL DO
STOP	stop the program
DELAY	pause for a certain length of time
WAIT	pause, until other simultaneous routines have completed their tasks
EXIT	stop a task, in a program that involves several tasks
HALT	interrupt the program, and free the terminal to do other tasks

These statements handle conditions:

STATEMENT'S FIRST WORD	WHAT THE COMPUTER WILL DO
IF	if a certain condition occurs now, do certain statements
ON	throughout the rest of the block of statements, if a certain unusual condition ever occurs (such as OVERFLOW), do certain statements
SIGNAL	pretend an unusual condition occurs (such as OVERFLOW), and do what the ON statement says
REVERT	throughout the rest of the block of statements, ignore the ON statements that occurred previously in the block

These statements handle variables:

STATEMENT'S FIRST WORD	WHAT THE COMPUTER WILL DO
DECLARE	make some variables be integers, others be reals, others be strings, others be filenames, etc.
DEFAULT	change the general assumptions about variables; for example, assume all variables are integers
ALLOCATE	create a temporary variable
FREE	destroy a temporary variable, so the memory it was using can be used for something else

These statements handle general logic:

STATEMENT'S FIRST WORD	WHAT THE COMPUTER WILL DO
GO	go to a different line
CALL	go to a subroutine
RETURN	return from a subprogram to the main routine
PROCEDURE	begin a program or subprogram
DO	begin a loop or compound statement
BEGIN	begin a block of statements
END	end a program, subprogram, loop, compound statement, or block of statement
ENTRY	when requested, skip the previous lines of the subprogram and begin here instead

Half of those statements are borrowed from FORTRAN, ALGOL, and COBOL.

from FORTRAN: FORMAT, STOP, CALL, RETURN, DO
from ALGOL: IF, GO, PROCEDURE, BEGIN, END
from COBOL: OPEN, CLOSE, READ, WRITE, DISPLAY, EXIT

Each PL/I statement ends with a semicolon. Besides the statements I've listed, you can also give an *assignment statement* (such as "N=5;"), a *null statement* (which consists of just a semicolon), and a *preprocessor statement* (which tells the computer how to create its own program).

EXTRA VOCABULARY

Of all the high-level computer languages, PL/I gives you the most control over the computer.

built-in functions, which you can put in an assignment statement, after the equal sign: ABS ACOS ADD ADDR ALL ALLOCATION ANY ASIN ATAN ATAND ATANH BINARY BIT BOOL CEIL CHAR CONJG COS COSD COSH COUNT DATAFIELD DATE DECIMAL DIM DIVIDE EMPTY ERF ERFC EXP FIXED FLOAT FLOOR HBOUND HIGH IMAG INDEX LBOUND LENGTH LINENO LOG LOG2 LOG10 LOW MAX MIN MOD MULTIPLY NULL OFFSET ONCODE ONCOUNT ONFILE ONKEY ONLOC POINTER POLY PRECISION PROD REPEAT ROUND SIGN SIN SIND SINH SQRT SUM TAN TAND TANH TIME TRANSLATE TRUNC VERIFY

pseudo-variables, which you can put in an assignment statement, before or after the equal sign: COMPLETION COMPLEX ONCHAR ONSOURCE PRIORITY REAL STATUS STRING SUBSTR UNSPEC

conditions, which you can put in the ON statement: AREA CHECK CONDITION CONVERSION ENDFILE ENDPAGE ERROR FINISH FIXEDOVERFLOW KEY NAME OVERFLOW PENDING RECORD SIZE STRINGRANGE STRINGSIZE SUBSCRIPTRANGE TRANSMIT UNDEFINEDFILE UNDERFLOW ZERODIVIDE

condition prefixes, which you can put before statements, such as assignment statements: NOCHECK NOCONVERSION NOFIXEDOVERFLOW NOOVERFLOW NOSIZE NOSTRINGRANGE NOSTRINGSIZE NOSUBSCRIPTRANGE NOUNDERFLOW NOZERODIVIDE

format items, which you can put in the FORMAT statement: A B C COLUMN E F LINE P PAGE R SKIP

attributes, which you can put in the DECLARE statement: ALIGNED AREA AUTOMATIC BACKWARDS BASED BINARY BIT BUFFERED BUILTIN CHARACTER COMPLEX CONDITION CONNECTED CONTROLLED DECIMAL DEFINED DIRECT ENTRY ENVIRONMENT EVENT EXCLUSIVE EXTERNAL FILE FIXED FLOAT GENERIC INITIAL INPUT INTERNAL IRREDUCIBLE KEYED LABEL LIKE OFFSET OPTIONS OUTPUT PICTURE POINTER POSITION PRINT REAL RECORD REDUCIBLE RETURNS SEQUENTIAL STATIC STREAM TASK TRANSIENT UNALIGNED UNBUFFERED UPDATE VARIABLE VARYING

environment options, which you can put in the DECLARE statement, in parentheses after the word ENVIRONMENT: ADDBUFF BLKSIZE BUFFERS COBOL CONSECUTIVE CTLASA CTL360 F FB FBS GENKEY INDEXAREA INDEXED KEYLENGTH KEYLOC LEAVE NCP NOWRITE RECSIZE REGIONAL REREAD SCALARVARYING TP TRKOFL U V VB VBS VERIFY VS

other words PL/I uses: ACTIVATE COPY DATA DEACTIVATE DESCRIPTORS EDIT ELSE FORTRAN FROM IGNORE IN INCLUDE INTER INTO KEYFROM KEYTO LINESIZE LIST MAIN NOMAP NOMAPIN NOMAPOUT NORESCAN ORDER PAGESIZE RANGE RECURSIVE REENTRANT REFER REORDER REPLY RESCAN SET SNAP SUB SYSIN SYSPRINT SYSTEM THEN TITLE TO VALUE WHEN WHILE

VERSIONS OF PL/I

I've been describing PL/I's fanciest versions (the "OS Optimizer" and the "Checkout Compiler"). If your computer isn't big enough to handle them, use a stripped-down version having fewer commands.

In 1972, Cornell University invented a pleasant stripped-down version called "PL/C", designed especially to help students debug their programs. In 1975, the University of Toronto invented an even smaller version called "SP/k". Although it allows fewer statements than PL/C, it runs faster and prints messages that are even more helpful. SP/k comes in several sizes: the tiniest is SP/1; the largest is SP/8.

COMMENTS BY EXPERTS

Here are comments on PL/I, by the top experts.

Saul Rosen

"Now (and on into the foreseeable future) most of the world's serious programs are written in FORTRAN and COBOL, not PL/I.

"I deplore the methods by which programming languages are developed. IBM designed PL/I as a crash project, to be completed in just a few months by a committee of six men, some of whom had other responsibilities at the same time. Considering the few people involved, and the little time they could devote to the project, their accomplishment was very impressive.

"Some people in IBM thought PL/I would gradually supplant FORTRAN, COBOL, and ALGOL and make those languages obsolete. PL/I contains most features of all those languages and provides many useful extras. With IBM's very enthusiastic backing after the design phase, it looked as if PL/I was bound to succeed. And it did succeed to some extent. PL/I is now used quite widely, and its use is increasing. Instead of replacing FORTRAN and COBOL, it seems on the way toward joining them as one of the standard production languages.

"Problems that can be handled well by FORTRAN can still be handled more efficiently in FORTRAN than in PL/I. When comparing PL/I to COBOL, some argue that COBOL is more natural and suitable for typical business applications.

"So far, PL/I has been almost exclusively an IBM language. Versions of PL/I on machines by other manufacturers are usually incomplete and inefficient. An exception is Honeywell, which inherited a rather good version of PL/I when it took over the General Electric computer division.

"The original descriptions of PL/I were informal. A group at the IBM research laboratory in Vienna tried to produce a more formal description; the resulting document was so thick it was called 'the Vienna telephone directory'."

Jean Sammet

"PL/I is the culmination of FORTRAN, ALGOL, and COBOL. It has included virtually all the good features from each of those languages. It has synthesized in a very reasonable way almost everything known about languages that solve scientific and business problems. But its attempt to *replace* FORTRAN, ALGOL, and COBOL has not yet succeeded.

"When people discuss computer languages, the most frequent arguments are about PL/I, ALGOL 68, and APL. The fact that most programming is done in languages *other* than these does not inhibit the debate about them. The public arguments about them come more from language developers and theoreticians than from the users, who remain rightly concerned about cost and compatibility and hence tend to stay with FORTRAN and COBOL."

Richard Conway & David Gries
the inventors of PL/C

"PL/I, PASCAL, and ALGOL are higher-level languages than FORTRAN. Turning a vague problem into a program requires extra effort if the target language is FORTRAN rather than PL/I.

"PL/I is, in most respects, less attractive than PASCAL or ALGOL. But it is also more widely used. In many universities this makes it the only alternative to FORTRAN or COBOL that is politically acceptable for an introductory course. PL/I is not easy to love; but if FORTRAN and COBOL are the only alternatives, one can learn to live with PL/I.

"PL/I has some unfortunate characteristics. Its notation is awkward and in many places inconsistent. PL/I also has a high astonishment factor—many features just don't work the way you would expect. For example, who would guess that 6E0 * 6E0 is 3E1?"

"Which language should be taught to beginning students?

"Although FORTRAN is the standard language for many applications, its clumsy features too often distract students from learning orderly methods of program construction.

"COBOL is another widely used language that must eventually be learned by many programmers. But its limitations make it a poor introductory language.

"BASIC is attractive because it is on many minicomputers and has an extremely simple notation. A student can learn BASIC more quickly than ALGOL. But he is then hampered by BASIC's short variable names when he tries to solve nontrivial programming problems.

"ALGOL 60, ALGOL W, and PASCAL are clean and elegant. But many teachers, students, and employers feel the advantages of a more widely accepted language outweigh any advantages of additional elegance.

"PL/I has the advantages that it includes reasonably good programming constructs and is widely used—though not as widely as FORTRAN or COBOL. So we chose to make our introductory language, SP/k, a subset of PL/I. The disadvantages of PL/I are that the full language includes peculiar and anomalous features and is expensive to process. These disadvantages can be overcome if PL/I is restricted to a subset whose size and elegance resembles ALGOL 60."

SIMPLE EXAMPLE

Suppose you survey 10 of your friends and ask each of them two questions:
1. In the next election, will you probably vote Republican or Democrat?
2. Are you male or female? (Maybe you can guess the answer by just looking at the person; but to be sure, you'd better ask.)

Suppose nobody gives an unusual answer (such as Prohibitionist or Communist or Transexual or Undecided). You think it would be cool to feed all the data into the computer. For example, if a person said "Republican Female", you'd feed the computer this line:

```
RF
```

If a person said "Democrat Male", you'd feed the computer this line:

```
DM
```

Use SPSS. Here's the SPSS program:

```
VARIABLE LIST  PARTY,SEX           Read each person's PARTY and SEX,
INPUT FORMAT   FIXED (2A1)           using this FORTRAN FORMAT:  "2A1".
N OF CASES     10                  There are 10 people.
INPUT MEDIUM   CARD                The data to read is on the "cards" below.
PRINT FORMATS  PARTY,SEX (A)       To print the PARTY and SEX, use "A" format.
CROSSTABS      TABLES=SEX BY PARTY Print a table showing how SEX relates to PARTY.
READ INPUT DATA                    The data to read is on the following lines.
RF
DM
RM
RM
DF                                 } the "data cards"
DM
DF
DF
RM
DF
FINISH                             The program is finished.
```

In the top line, the word PARTY begins in column 16. Most SPSS statements consist of a *control field* (columns 1-15) followed by a *specification field* (columns 16-80).

When you run the program, the computer will print this kind of table:

	R	D	ROW TOTAL
M	3	2	5
	60.0%	40.0%	50.0%
	75.0%	33.3%	
	30.0%	20.0%	
F	1	4	5
	20.0%	80.0%	50.0%
	25.0%	66.7%	
	10.0%	40.0%	
COLUMN TOTAL	4 40.0%	6 60.0%	10 100.0%

Look at the top number in each box. Those numbers say there were 3 male Republicans, 2 male Democrats, 1 female Republican, and 4 female Democrats. The first box says: the 3 male Republicans were 60% of the males, 75% of the Republicans, and 30% of the total population.

The computer prints the table in reverse-alphabetical order: "M" before "F", and "R" before "D". Each row is a SEX, and each column is a PARTY. In the program, if you change "SEX BY PARTY" to "PARTY BY SEX", each row will be a PARTY, and each column will be a SEX.

FANCY FEATURES

The CROSSTABS statement has *options*. Here are some of them.

option 3: don't print the row percentages (the 60.0%, 40.0%, 20.0%, and 80.0%)
option 4: don't print the column percentages (the 75.0%, 33.3%, 25.0%, and 66.7%)
option 5: don't print the total percentages (the 30.0%, 20.0%, 10.0%, and 40.0%)

If you want options 3 and 5, insert this statement underneath the CROSSTABS statement:

```
OPTIONS       3,5
```

The CROSSTABS statement has *statistics*. Here are some of them:

1. chi-square, its degrees of freedom, and its level of significance
2. phi or Cramer's V
3. contingency coefficient
4. lambda, symmetric and asymmetric
5. uncertainty coefficient, symmetric and asymmetric
6. Kendall's tau b and its level of significance
7. Kendall's tau c and its level of significance
8. gamma
9. Somer's D

Those statistics are numbers that help you analyze the crosstab table. If you want statistics 1 and 8, insert this statement underneath the CROSSTABS and OPTIONS statements:

```
STATISTICS    1,8
```

It makes the computer print statistics 1 and 8 underneath the table. If you want the computer to print all 9 statistics, say:

```
STATISTICS    ALL
```

The CROSSTABS statement is called a *procedure*. Here is a complete list of procedures SPSS can handle:

AGGREGATE ANOVA BREAKDOWN CANCORR CONDESCRIPTIVE CROSSTABS DISCRIMINANT
FACTOR FREQUENCIES GUTTMAN SCALE NONPAR CORR ONEWAY PARTIAL CORR
PEARSON CORR REGRESSION SCATTERGRAM T-TEST WRITE CASES

Each procedure has its own OPTIONS and STATISTICS.

SPSS includes many other kinds of statements:

ADD CASES ADD DATA LIST ADD SUBFILES ADD VARIABLES ALLOCATE ASSIGN MISSING
COMMENT COMPUTE COUNT DATA LIST DELETE SUBFILES DELETE VARS DO REPEAT
DOCUMENT EDIT END REPEAT FILE NAME GET ARCHIVE GET FILE IF KEEP VARS
LIST ARCHINFO LIST CASES LIST FILEINFO MERGE FILES MISSING VALUES NUMBERED
PAGESIZE PRINT BACK RAW OUTPUT UNIT READ MATRIX RECODE REORDER VARS
RUN NAME RUN SUBFILES SAMPLE SAVE ARCHIVE SAVE FILE SELECT IF SORT CASES
SUBFILE LIST TASK NAME VALUE LABELS WEIGHT WRITE FILEINFO

SPSS contains more statistical features than any other language. If you don't need quite so many features, use an easier language, such as STATPAK, DATATEXT, or IMPRESS.

LOGO

LOGO began in 1967, during an evening at Dan Bobrow's home in Belmont, Massachusetts.

Dan had gotten his Ph.D. from MIT and was working for a company called *Bolt, Beranek, and Newman (BBN)*. In Dan's living room were three of his colleagues from BBN (Wally Feurzeig, Cynthia Solomon, and Dick Grant) and an MIT professor: Seymour Papert.

BBN had been trying to teach young kids how to program by using BBN's own language (TELCOMP), which was a variation of JOSS. BBN had asked Professor Seymour Papert for his opinion. The group was all gathered in Dan's house to hear Seymour's opinion.

Seymour chatted with the group, and the entire group agreed with Seymour on several points. First, TELCOMP was *not* a great language for kids. It placed too much emphasis on mathematical formulas. The group agreed that instead of struggling with math, the kids ought to be having more fun by programming the computer to handle strings instead.

The group also agreed that the most sophisticated language for handling strings was LISP. But the group also agreed that LISP was too complex for kids.

The group concluded that a new, simplified version of LISP ought to be invented, especially for kids. The group called the new, simplified version "LOGO". That's how LOGO began. Professor Seymour Papert was the guiding light, and all the other members of the group gave helpful input during the conversation.

That night, after the guests had left, Dan Bobrow went to the terminal in his bedroom, and started programming the computer to understand LOGO. Specifically, he wrote a LISP program that explained to the computer how to handle LOGO. And that's how LOGO was born.

Work on LOGO continued. The three main researchers who continued improving LOGO were Seymour (the MIT guru), Wally (from BBN), and Cynthia (also from BBN). LOGO looked something like "LISP without parentheses".

After helping BBN for a year, Seymour returned to MIT. Cynthia and several other BBN folks went with him. In MIT's Artificial Intelligence Laboratory, Seymour and his friends continued to improve LOGO.

During the first few years, LOGO was like most other computer languages: it was abstract and boring. But in the spring of 1970, a surprising creature walked into the LOGO lab. It was a big yellow mechanical turtle. It looked like "half a grapefruit on wheels". And it had a pen in its belly:

wheel pen wheel

It also had a horn, feelers, and several other fancy attachments. To use the turtle, you put paper all over the floor and then programmed the turtle to roll across the paper. As the turtle rolled, the pen in its belly drew pictures on the paper. The turtle was controlled remotely by a big computer, which was programmed in...LOGO!

Suddenly, LOGO had become fun. LOGO had become a language whose main purpose was to control the turtle. Kids who watched the turtle screamed with delight and wanted to learn how to program it, by using LOGO. So LOGO had suddenly become a favorite programming game for kids. Even teeny-weeny kids, who were just 7 years old, started programming in LOGO. Those kids were barely old enough to read. But reading and writing were *not* prerequisites for learning how to program in LOGO. All you had to know was that "FD 3" made the turtle go forward 3 steps, and "RT 30" made the turtle turn to the right 30 degrees.

As for the rest of LOGO— all that abstract stuff about strings and numbers and LISP-like lists— the kids ignored it. They wanted to use just the commands "FD" and "RT" that moved the turtle.

The U.S. Government's National Science Foundation donated money to MIT, so that MIT could continue improving LOGO further. Many kids came into the LOGO lab to play with the turtles.

But the turtles were expensive, and so were the big computers that controlled them. If all the kids in the United States were to get a chance to play with LOGO, the first problem was to reduce the cost of the turtle and its controlling computer.

During the early 1970's, the cost of video display terminals decreased dramatically. Eventually, MIT decided that instead of building a mechanical turtle, it was cheaper to buy a video display terminal whose screen could display a picture of a turtle. So MIT stopped using mechanical turtles, and started using graphics terminals instead.

During the 1970's, computers became cheaper too. The original version of LOGO was done on BBN's expensive weird computer (the MTS 940). Later versions were done on the PDP-1 (in 1968), then the PDP-10 (in 1970), and finally on a cheaper computer: the PDP-11 minicomputer (in 1972).

At the end of the 1970's, microcomputers were invented, such as the Apple and the Radio Shack. MIT wanted to put LOGO on those popular microcomputers. But MIT was running out of money. Who would pay for the research to put LOGO on a microcomputer?

Texas Instruments (TI) came to the rescue. TI agreed to pay MIT for the research to put LOGO on TI's microcomputers (the TI-99/4 and the TI-99/4A).

TI and MIT thought the job would be easy, since MIT had already written a PASCAL program that made the computer understand LOGO, and since TI had already written a version of PASCAL for the CPU chip inside the TI-99/4. Initially, MIT was worried, because the PASCAL program running on MIT's PDP-10 computer handled LOGO too slowly; but TI claimed TI's PASCAL was faster than the PDP-10's and that LOGO would therefore run fast enough on the TI.

TI was wrong. TI's PASCAL couldn't make LOGO run fast enough; and TI's PASCAL also required too much RAM. So TI had to laboriously take MIT's research (on the PDP-10) and laboriously translate it into TI's assembly language, by hand.

The hand translation went slower than TI expected. TI became impatient, and wanted to start selling LOGO soon. So TI took a short-cut: it omitted parts of LOGO. The main part of LOGO that TI omitted was decimals: TI's LOGO understands just integers.

After TI started selling "TI LOGO", the MIT group invented a version of LOGO for the Apple. The Apple version includes decimals (unlike the TI version). But on the other hand, the Apple version omits "sprites" (which are animated creatures that carry objects across the screen), because Apple's hardware can't handle sprites quickly enough. (The TI's hardware is fancier and *does* handle sprites.)

MIT wanted to sell the Apple version to schools, since most schools own Apples instead of TI computers. But if MIT were to start selling a disk containing "MIT LOGO for the Apple" and make lots of money, MIT might have sunk into legal trouble, since MIT was supposed to be non-profit. And anyway, who "owned" LOGO? Possible contenders were:

MIT, which had done most of the research
BBN, which had already trademarked the name "LOGO" and had done the early research
the U.S. Goverment, whose National Science Foundation had paid for much research
TI, which had also paid for much research

Eventually, MIT solved the legal problems, and sold the rights to "MIT Apple LOGO". The buyer was Terrapin, a company that had already been manufacturing mechanical turtles.

Terrapin hired some MIT graduates, to help Terrapin improve MIT Apple LOGO. Today, you can buy the improved MIT Apple LOGO from Terrapin for $150. Contact Terrapin at 380 Green St., Cambridge, MA 02139, 617-492-8816. Quantity discounts are available to dealers and schools. To use Terrapin's LOGO, you need an Apple that has a disk drive and 64K of RAM. (To get 64K of RAM, either buy a 64K Apple 2e or add a 16K RAM card to a 48K Apple 2-plus.)

Another company, called "Krell", also obtained the rights to MIT Apple LOGO. The Krell version of MIT Apple LOGO costs $180 (which is more than Terrapin) and contains fewer improvements.

Back when MIT was consulting its lawyers about who owned Apple LOGO, a group of MIT's faculty and students became impatient. The group, headed by Cynthia Solomon (one of the original inventors of LOGO), left MIT and formed a company called *LOGO Computer Systems Incorporated (LCSI)*. That company invented its own version of LOGO for the Apple. Like the MIT-Terrapin-Krell versions, it requires a disk drive and 64K of RAM; but it's newer and costs $175. Apple Computer Company is distributing the LCSI version to all Apple dealers.

The TI and MIT-Terrapin-Krell versions were all derived from the version that MIT wrote in PASCAL on the PDP-10. The LCSI version was derived from the version that LCSI wrote in LISP on a special "LISP minicomputer".

Which is better: Terrapin or LCSI? There's no simple answer. Each version contains features that the other version lacks. If you ask one of those companies about a feature that only the opposite company has, whichever company you're asking will say defensively "We're working on that feature too, and we'll have it soon" or "That feature isn't important, and just wastes space in the 64K RAM".

At the moment, LCSI is better at handling disks (especially random-access data files), while Terrapin is better at handling RAM (especially lists stored in the RAM). LCSI is slightly better at handling graphics (especially when the turtle tries to go off the edge of the screen), whereas Terrapin is slightly better at handling music (since Terrapin includes a demonstration music program). Each company claims that its LOGO works "faster" than the other company's. (One of them must be lying.) Each company claims that its version of the the IF...THEN...ELSE statement is easier to understand than the other company's. (Try them both, and see which version of IF...THEN...ELSE *you* think is the easiest!) Terrapin's editor (which helps you edit your program) is a bit easier to learn than LCSI's; on the other hand, LCSI lets beginners *avoid* the editor more easily.

Terrapin is a small company in Cambridge, Massachusetts. LCSI is larger and based in Montreal.

Recently, Terrapin and LCSI have invented even better versions of LOGO for other computers. Terrapin invented the wonderful LOGO sold by Commodore for the Commodore 64; LCSI invented the wonderful LOGO sold by Atari and IBM.

Many of LOGO's designers *hate* BASIC, and believe that BASIC should be eliminated from schools altogether. They believe that LOGO is easier to learn than BASIC, and that LOGO encourages a kid to be more creative. They also believe that LOGO leads the kid to think in a more organized fashion than BASIC. They also argue that since LOGO is best for little kids, and since switching languages is difficult, the kids should continue using LOGO until they graduate from high school and should never use BASIC. That argument is wrong: it ignores the fact that a knowledge of BASIC is *essential* to surviving in our computerized society. Today, most programs are still written in BASIC, not LOGO, because BASIC consumes less RAM and because the newest versions of BASIC contain many practical features for business and science and graphics that LOGO lacks.

One of the nicest things about LOGO is that you can change it, and turn it into your *own* language! That's because LOGO lets you invent your own commands and add them to the LOGO language. A language (such as LOGO) that lets you invent your own commands is called an *extensible language*. Although some of the classical languages (such as LISP) are extensible also, LOGO is *more* extensible and also pleasanter.

FORTH

Like LOGO, FORTH is extensible. And FORTH has two big advantages over LOGO: FORTH consumes less memory and runs faster. You can easily run FORTH on a machine that has only 8K of RAM, and the computer handles FORTH almost as fast as it handles assembly language. Because FORTH is extensible and consumes so little of the computer's memory and time, professional programmers often use it. For example, EASYWRITER (which is a word processing program for the Apple and the IBM Personal Computer) was written in FORTH. (Unfortunately, EASYWRITER contains several bugs, but that's because EASYWRITER's programmers weren't very bright.)

In FORTH, if you want to add 2 and 3 (to get 5), you do *not* type 2+3. Instead, you must type 2 3 +. The idea of putting the plus sign afterwards (instead of in the middle) is called *postfix notation*. The postfix notation (2 3 +) has two advantages over infix notation (2+3): the computer can handle postfix notation more quickly, and you never need to use parentheses for "order of operations". On the other hand, postfix notation seems inhuman: it's hard for a human to read. Because it's hard for humans, FORTH is *not* a good language for little children. On the other hand, if you're a professional programmer who wants to make your programs run super quickly and eat up hardly any RAM, use FORTH.

Like FORTH, Hewlett-Packard pocket calculators use postfix notation. So if you've already had experience with a Hewlett-Packard calculator, you'll find FORTH easy.

Postfix notation is the reverse of *prefix notation* (+ 2 3), which was invented around 1926 by the Polish mathematician Lukasiewicz. So postfix notation is often called *reverse Polish notation*.

Because FORTH is so difficult for a human to read, cynics call it "an inhuman Polish joke".

FORTH was invented by Chuck Moore, during his spare time while he worked at many schools and companies. He wanted to name it "FOURTH", because he considered it to be an ultra-modern "fourth-generation" language; but he was using an old IBM 1130 minicomputer, which couldn't handle a name as long as "FOURTH"; so he omitted the letter "U".

PILOT

PILOT was invented at the San Francisco branch of the University of California, by John Starkweather in 1968. It's easier to learn than BASIC; but it's supposed to be used by teachers instead of students. By using PILOT, teachers can easily program the computer to tutor students about history, geography, math, French, and other schoolbook subjects.

For example, suppose you're a teacher and want to make the computer chat with your students. Here's how to do it in BASIC, and more easily in PILOT:

BASIC program

```
10 PRINT "I AM A COMPUTER"
20 INPUT "DO YOU LIKE COMPUTERS";A$
30 IF A$="YES" OR A$="YEAH" OR A$="YEP" OR A$="SURE" OR A$="SURELY" OR A$="I SURE DO"
   THEN PRINT "I LIKE YOU TOO" ELSE PRINT "TOUGH LUCK"
```

PILOT program

```
T:I AM A COMPUTER          Type "I AM A COMPUTER".
T:DO YOU LIKE COMPUTERS?   Type "DO YOU LIKE COMPUTERS?"
A:                         Accept the human's answer.
M:YE,SURE                  Match. (See whether the answer contains "YE" or "SURE".)
TY:I LIKE YOU TOO          If there was a match, type "I LIKE YOU TOO".
TN:TOUGH LUCK              If no match, type "TOUGH LUCK".
```

Notice that the PILOT program is briefer than BASIC.

Atari, Apple, and Radio Shack all sell versions of PILOT that include commands to handle graphics. Atari's version is better than Apple's and Radio Shack's, because Atari's includes the fanciest graphics and music and even a LOGO-like turtle, and because it's also the easest version to learn how to use.

Unfortunately, few teachers use PILOT. Although PILOT is easier than BASIC, most teachers prefer to learn BASIC, because BASIC is available on more computers and costs less and accomplishes a greater variety of tasks. The teachers don't want to spend their time and money to learn a second language: they stick with just BASIC.

PASCAL

In 1968, a European committee tried to invent an improved version of ALGOL. Most of the committee's members agreed with each other and produced a version called "ALGOL 68". But a few of the committee's members were dissidents who thought the rest of the committee was crazy. One of the dissidents was Niklaus Wirth. He quit the committee and created his own version of ALGOL. He called his version "PASCAL". Today, most programmers feel that Niklaus Wirth was right and the rest of the committee was wrong: PASCAL is better than ALGOL 68.

Niklaus Wirth wrote PASCAL in Switzerland, for a CDC maxicomputer that used cards. His version of PASCAL couldn't handle terminals; it couldn't handle random-access data files; and it couldn't handle strings well. Those three limitations were corrected in later versions of PASCAL— especially the version invented at the University of California at San Diego (UCSD), which even includes LOGO-style commands that move a turtle.

Apple Computer Company obtained permission to sell an Apple version of UCSD PASCAL. Apple ran full-page advertisements, bragging that the Apple computer could handle PASCAL and that other microcomputers could not.

For $495, Apple Computer Company would sell you the "Apple Language System", which included 4 disks containing PASCAL, 2 disks containing souped-up BASIC, and a card containing 16K of extra RAM. Many people spent the $495 for PASCAL.

Those people were disappointed. They expected that by spending $495, they'd be able to write programs more easily. But PASCAL is no easier than BASIC. In fact, PASCAL is more *difficult* to learn than BASIC.

PASCAL is helpful only if the program you're writing is very, very long. PASCAL helps you organize and dissect long programs more easily than BASIC. But the average Apple owner never writes long programs, and therefore never needs PASCAL.

Many customers felt "ripped off": they had spent $495 and received no benefit in return. But maybe that's what "marketing" is all about.

Many programmers who've been writing large FORTRAN programs for large computers are switching to PASCAL, because PASCAL helps you organize large programs better, and because FORTRAN is archaic. Many programmers who've been using PL/I are switching to PASCAL, because PASCAL consumes less RAM than PL/I and can therefore fit in smaller computers.

Today, the most powerful version of PASCAL for microcomputers is *Turbo PASCAL*. It runs on the IBM PC and CP/M computers, costs about $50, and is published by Borland International (4807 Scotts Valley Dr., Scotts Valley, CA 95066, phone 408-438-8400 or 800-227-2400 extension 968).

Beginners might prefer *PASCAL 80* instead, because its editor is easier to learn. It's published by New Classics Software (239 Fox Hill Rd., Denville, NJ 07834, phone 201-625-8838). It's available for CP/M computers ($39) and for Radio Shack computers ($79 for the full version, $15 for a stripped-down version, $279 for unlimited reproduction of the full version throughout a school building).

C

Many programmers are starting to use a new language called "C". It was invented at Bell Telephone Laboratories by Dennis Ritchie. It's a slight improvement over an earlier language, which was called "B".

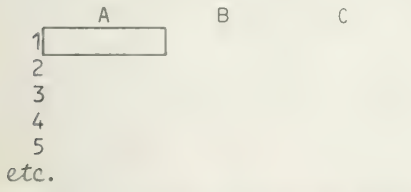
By using C, programmers made the PDP-11 act in a nice, strange way. That way is called UNIX. It's an operating system, written in C. Today, the C language and the UNIX operating system are available on many minicomputers and microcomputers. At Radio Shack's headquarters, the staff is writing programs for Radio Shack's computers by using C.

C is popular because it looks like most other structured languages (ALGOL, PASCAL, and PL/I) yet consumes hardly any RAM; and the computer handles it very quickly.

VISICALC

VISICALC is one of the most important computer languages ever invented. Here's how to use it....

Turn your computer *off*. Put the Visicalc disk into the disk drive, and close the disk drive's door. Turn your computer *on*. After a short delay, you'll see this on your screen:



etc.

That means the computer's waiting for you to fill in a table of numbers. The table's first column is labeled A; the second column is labeled B; the third column is labeled C; etc. The table's rows are labeled 1, 2, 3, 4, 5, etc. A box appears at the table's upper-left corner, which is called position A1 (since it's in column A and row 1).

On your keyboard, you'll see a key that has a right arrow. If you press it, the screen's box moves to the right, so that it arrives at position B1. Each time you press that right-arrow key, the box moves further to the right. Similarly, each time you press the *left*-arrow key, the box moves further to the *left*.

If your computer's keyboard is modern, one of the keys has an arrow that points down. If you press that key, the screen's box moves down. Similarly, if you press the key that has an *up*-arrow, the screen's box moves back up. (Unfortunately, the Apple's keyboard is old-fashioned, and doesn't have an up-arrow or down-arrow. So to do Visicalc easily, choose a computer that's not an Apple.)

By using the arrow keys— right, left, down, and up— you can move the box anywhere on the screen.

Move the box wherever you please, type a number (such as 4217.95), and press the return key. The number will appear where the box was. Then move the box, type another number, and press the return key again. Do that several times— so that the table starts filling up with numbers.

Suppose you've typed a number at position A1 and typed another number at position B1. Suppose you want the computer to add those two numbers together and put the sum at position C1. Here's how: move the box to position C1, then type the formula +A1+B1. When you press the return key, the number that's the sum of A1 and B1 immediately appears in the box C1.

That formula— +A1+B1— told the computer that C1 should be the sum of A1 and B1, *forever*. If you move the box back to position A1 and change the number in position A1, the computer will automatically change the number in position C1, so that C1 is still the sum of A1 and B1.

If you're an accountant— or wish you were— you'll love Visicalc, because each time you change a number in your table, Visicalc makes the computer automatically change the totals and subtotals.

Besides finding sums, Visicalc also lets you subtract, multiply, divide, and even do exponents and trigonometry. It also lets you put a practical label on each row and column; for example, you can let the bottom row be labeled "total net profit", and you can let the rightmost column be labeled "percent increase over last year". It will also copy the table from your screen to your disk and to your printer.

Visicalc is like a word processor, except that it's mainly for editing *tables of numbers* instead of words. So it ought to be called a "table processor".

It's easy to understand. (If you're too stupid to understand Visicalc, you're called a "Visiklutz".)

Use Visicalc to explore new strategies for making your business profitable. If you experiment with a new strategy— by changing the numbers in the tables – the computer will immediately tell you the new net profit that results.

Visicalc can keep track of your company's inventory. Many businesses use Visicalc instead of buying an "inventory program".

Visicalc was invented because Dan Bricklin, a graduate student at the Harvard Business School, had to finish a project that required accounting, and found that his pocket calculator worked too slowly. He asked a friendly programmer, Bob Frankston, to help. They invented Visicalc— Dan did the designing, and Bob did the programming. Dan showed it to his professor, who said, "It will never sell; but if you don't believe me, ask Dan Fylstra." So Dan Bricklin asked Dan Fylstra, who ran a software house called Personal Software. Dan bought Visicalc, and within 2½ years sold about 150,000 copies retailing at about $150, making a total retail sale of about $22,500,000. Moral: don't believe Harvard professors. All three of the people responsible— Dan Bricklin, Bob Frankston, and Dan Fylstra— profited enormously.

Today, Visicalc is available for most microcomputers: Apple, Radio Shack, Commodore, Atari, Hewlett-Packard, and the IBM PC. But it is *not* available for CP/M computers. If you have a CP/M computer, you can try *Supercalc* instead, which resembles Visicalc and is published by Sorcim, a cute company whose name is "Micros" spelled backwards!

If you have an IBM PC, you'll prefer *1-2-3*, which is fancier than Visicalc and is published by Lotus, a cute company headed by a Transcendental Meditation instructor.

ADA

In 1975, the U.S. Department of Defense decided to invent a new computer language. But before inventing a new language, the Department had to decide what *kind* of language it wanted. So the Department wrote a list of requirements the language would have to meet.

That list of requirements was improved several times. The first version of the list was called the Strawman Requirements (1975); then came improved versions, called Woodenman (1975), Tinman (1976), Ironman (1978), and finally Steelman (1979).

While the Department was moving from Strawman to Steelman, it also checked whether any existing computer language could meet such requirements. The Department concluded that no existing computer language came even close to meeting the requirements, and so a new language would indeed have to be invented. The Department also concluded that the new language would have to resemble PASCAL, ALGOL 68, or PL/I, but be better.

The Department decided to hold a contest, to see who could invent the new language. 16 companies entered the contest: they invented 16 languages that they hoped would meet the Steelman Requirements. The Department selected 4 semifinalists, and told them to continue their research and try harder.

The semifinalist companies were CII-Honeywell-Bull (which is French and owned partly by Honeywell), Intermetrics (in Cambridge, Massachusetts), SRI International, and Softech.

Several months later, those 4 semifinalists submitted improved designs. All the improved designs were souped-up versions of PASCAL (instead of ALGOL 68 or PL/I). To make the contest fair and prevent bribery, the judges weren't told which design belonged to which company; the 4 were called "Green", "Red", "Yellow", and "Blue".

Yellow and Blue lost. Green and Red became the finalists. The Department told Green's inventor (which was CII-Honeywell-Bull) and Red's inventor (Intermetrics) to continue their research and try even harder.

In 1979, CII-Honeywell-Bull (Green) and Intermetrics (Red) submitted their improved versions. Green won. The Department decided that the Green language would be called "ADA", to honor Ada Lovelace, the woman who was the world's first programmer. (For the crazy story of Ada Lovelace's life, read pages 56-57 of volume 4.)

So ADA is a PASCAL-like language developed by a French company under contract to the U.S. Department of Defense.

Will ADA become popular? Wait and see. Many researchers are trying to make computers understand ADA. After computers understand ADA and programmers try to learn that new language, we'll know whether that language is a pleasure or a pain.

DBASE 2

DBASE 2 is a language that lets you easily handle data files on your disk. It organizes the data by using a very flexible and wonderful method, called a "relational data base". It's an improvement of an earlier language, called VULCAN.

It lists for $700, but most discount dealers sell it for about $380. It works on the IBM PC and on CP/M computers.

EASY

EASY is a language I've been developing. It combines the best features of all other languages. It's easy to learn, because it uses just these twelve keywords:

SAY and GET and LET
REPEAT and SKIP and HERE
IF and PICK and LOOP
PREPARE and DATA and HOW

Let's look at them.

SAY. EASY uses the word SAY instead of BASIC's word PRINT, because SAY is briefer to type. For example, if you want the computer to say the answer to 2+2, give this command:

```
SAY 2+2
```

The computer will say the answer:

```
4
```

Whenever the computer prints, it automatically prints a blank space afterwards, and does *not* automatically press the return key. So if you run this program—

```
SAY "LOVE"
SAY "HATE"
```

the computer will say this:

```
LOVE HATE
```

Here's a fancier example:

```
SAY "LOVE" AS 3 AT 20 15 TRIM !
```

The "AS 3" is a format: it makes the computer print just the first 3 letters of LOVE. The "AT 20 15" makes the computer begin printing LOVE at the screen's pixel whose X coordinate is 20 and whose Y coordinate is 15. Normally, the computer prints a blank space after everything, but the word TRIM suppresses that blank space. The exclamation point makes the computer press the return key afterwards. Here's another example:

```
SAY TO SCREEN PRINTER HARRY
```

It means that henceforth, whenever you give a SAY command, the computer will print the answer simultaneously onto your screen, onto your printer, and onto a disk file named HARRY. If you ever want to cancel that "SAY TO" command, give a "SAY TO" command that contradicts it.

GET. EASY uses the word GET instead of BASIC's word INPUT, because GET is briefer to type. The command GET X makes the computer wait for you to input a value of X. Normally, the GET command is preceded by a SAY command that makes the computer ask a question. You can make the GET command fancy, like this:

```
GET X AS 3 AT 20 15 WAIT 5
```

The "AS 3" tells the computer that X will be just 3 characters; the computer waits for you to type just 3 characters and doesn't require you to press the return key afterwards. The "AT 20 15" makes the computer move to pixel 20 15 before your typing begins, so your input appears at that part of the screen. The "WAIT 5" makes the computer wait just 5 seconds for your response. It sets TIME equal to the number of seconds you took to reply, unless you took too long, in which case it sets TIME equal to -1.

LET. The LET statement resembles BASIC's. For example, you can say:

```
LET R=4
```

To let R be a random decimal, type:

```
LET R=RANDOM
```

To let R be a random integer from 1 to 6, type:

```
LET R=RANDOM TO 6
```

To let R be a random integer from -3 to 5, type:

```
LET R=RANDOM FROM -3 TO 5
```

REPEAT. If you put the word REPEAT at the bottom of your program, the computer will repeat the entire program again and again, forming an infinite loop.

SKIP. If you put the word SKIP in the middle of your program, the computer will skip the bottom part of the program. SKIP is like BASIC's STOP or END.

HERE. In the middle of your program, you can say:

```
HERE IS FRED
```

An earlier line can say SKIP TO FRED; a later line can say REPEAT FROM FRED. The SKIP TO and REPEAT FROM are like BASIC's GO TO.

IF. In your program, a line can say:

```
IF X<3
```

Underneath that line, you must put some indented lines. The computer will do the indented lines, if X<3. Problem: suppose a student's score is from 0 to 100; if the student's score is 100, make the computer say "PERFECT"; if the score is below 100 but at least 70, make the computer say the score and also say "OKAY THOUGH NOT PERFECT"; if the score is below 70, make the computer say "YOU FAILED". Here's how:

```
IF SCORE=100
  SAY "PERFECT"
IF SCORE<100 AND SCORE>=70
  SAY SCORE
  SAY "OKAY THOUGH NOT PERFECT"
IF SCORE<70
  SAY "YOU FAILED"
```

To shorten the program, use the words NOT and BUT:

```
IF SCORE=100
  SAY "PERFECT"
IF NOT BUT SCORE>=70
  SAY SCORE
  SAY "OKAY THOUGH NOT PERFECT"
IF NOT
  SAY "YOU FAILED"
```

The phrase "IF NOT" is like BASIC's ELSE. The phrase "IF NOT BUT" is like BASIC's ELSE IF.

PICK. You can shorten that example even further, by telling the computer to pick just the first IF that's true:

```
PICK SCORE
  IF 100
    SAY "PERFECT"
  IF >=70
    SAY SCORE
    SAY "OKAY THOUGH NOT PERFECT"
  IF NOT
    SAY "YOU FAILED"
```

LOOP. In your program, you can say LOOP. It means: do the lines that are indented underneath, repeatedly. This program makes the computer say the words CAT and DOG repeatedly:

```
LOOP
  SAY "CAT"
  SAY "DOG"
```

This program makes the computer say 5, 8, 11, 14, and 17:

```
LOOP I FROM 5 BY 3 TO 17
  SAY I
```

That LOOP statement is like BASIC's "FOR I = 5 TO 17 STEP 3". If you omit the "BY 3", the computer will assume "BY 1". If you omit the "FROM 5", the computer will assume "FROM 1". If you omit the "TO 17", the computer will assume "to infinity". To make the computer count down instead of up, insert the word DOWN, like this:

```
LOOP I FROM 17 DOWN BY 3 TO 5
```

PREPARE. If you want the computer to do something unusual, you should tell the computer to PREPARE for it. For example, if you want to use subscripted variables such as X(100), you should tell the computer:

```
PREPARE X(100)
```

In that example, PREPARE is like BASIC's DIM.

DATA. EASY's DATA statement resembles BASIC's. But instead of saying READ X, you say:

```
LET X=NEXT
```

HOW. In EASY, you can give any command you wish, such as:

```
PRETEND YOU ARE HUMAN
```

If you give that command, you must also give an explanation that begins with these words:

```
HOW TO PRETEND YOU ARE HUMAN
```

Interrelated features. In the middle of a loop, you can abort the loop. To skip out of the loop (and progress to the rest of the program), say SKIP LOOP. To hop back to the beginning of the loop (to do the next iteration of the loop), say REPEAT LOOP. Similarly, you can say SKIP IF (which makes the computer skip out of an IF) and REPEAT IF (which makes the computer repeat the IF statement, and thereby imitate PASCAL's WHILE).

Apostrophe. The computer ignores everything that's to the right of an apostrophe (unless the apostrophe is between quotation marks or in a DATA statement). So to put a comment into your program, begin the comment with an apostrophe. (EASY uses the apostrophe in roughly the same way as extended Microsoft BASIC.)

Comma. If two statements begin with the same keyword, you can combine them into a single statement, by using a comma. For example, instead of saying—

```
LET X=4
LET Y=7
```

you can say:

```
LET X=4, Y=7
```

And instead of saying—

```
PRETEND YOU ARE HUMAN
PRETEND GOD IS DEAD
```

you can say:

```
PRETEND YOU ARE HUMAN, GOD IS DEAD
```

More info. For details and extended features, phone me at 617-266-8128.

CHAPTER 8

BIT FIDDLING

NUMBER SYSTEMS

COMPUTERS AREN'T HUMAN

Most humans use the *decimal system*, which consists of ten digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), because humans have ten fingers. The computer does not have fingers, so it prefers other number systems instead. Here they are....

BINARY

Look at these powers of 2:

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$

Now try an experiment: pick your favorite positive integer, and try to write it as a sum of powers of 2. For example, suppose you pick 45. You can write it as 32+8+4+1. Suppose you pick 74; you can write it as 64+8+2. Suppose you pick 77; you can write it as 64+8+4+1. *Every* positive integer can be written as a sum of powers of 2.

Let's put those examples in a table:

ORIGINAL NUMBER	WRITTEN AS A SUM OF POWERS OF 2	DOES THE SUM CONTAIN... 64?	32?	16?	8?	4?	2?	1?
45	32+8+4+1	no	yes	no	yes	yes	no	yes
74	64+8+2	yes	no	no	yes	no	yes	no
77	64+8+4+1	yes	no	no	yes	yes	no	yes

To write those numbers in the *binary system*, replace "no" by 0 and "yes" by 1:

DECIMAL SYSTEM	BINARY SYSTEM
45	0101101 (or simply 101101)
74	1001010
77	1001101

The *decimal system* uses the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 and uses these columns:

thousands hundreds tens units

For example, the decimal number 7105 means "7 thousands + 1 hundred + 0 tens + 5 units

The *binary system* uses only the digits 0 and 1, and uses these columns:

sixty-fours thirty-twos sixteens eights fours twos units

For example, the binary number 1001101 means "1 sixty-four + 0 thirty-twos + 0 sixteen 1 eight + 1 four + 0 twos + 1 unit"; in other words, it means seventy-seven.

In elementary school, you were taught how to do arithmetic in the decimal system. You had to memorize the addition and multiplication tables:

DECIMAL ADDITION

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

DECIMAL MULTIPLICATION

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

In the binary system, the only digits are 0 and 1, so the tables are briefer:

BINARY ADDITION

	0	1
0	0	1
1	1	10

← *because two is written "10" in binary*

BINARY MULTIPLICATION

	0	1
0	0	0
1	0	1

If society had adopted the binary system instead of the decimal system, you would have been spared many hours of memorization!

Usually, when you ask the computer to perform a computation, it converts your numbers from the decimal system to the binary system, performs the computation by using the binary addition and multiplication tables, and then converts the answer from the binary system to the decimal system, so you can read it. For example, if you ask the computer to print 45+74, it will do this:

```
 45 converted to binary is   101101
+74 converted to binary is +1001010
                            1110111   converted to decimal is 119
```

because 1+1=10

The conversion from decimal to binary and then back to decimal is slow. But the computation itself (in this case, addition) is quick, since the binary addition table is so simple. The only times the computer must convert is during input (decimal to binary) and output (binary to decimal). The rest of the execution is performed quickly, entirely in binary.

You know fractions can be written in the decimal system, by using these columns:

units point tenths hundredths thousandths

For example, $1\frac{5}{8}$ can be written as 1.625, which means "1 unit + 6 tenths + 2 hundredths + 5 thousandths".

To write fractions in the binary system, use these columns instead:

units point halves fourths eighths

For example, $1\frac{5}{8}$ is written in binary as 1.101, which means "1 unit + 1 half + 0 fourths + 1 eighth".

You know $\frac{1}{3}$ is written in the decimal system as 0.3333333..., which unfortunately never terminates. In the binary system, the situation is no better: $\frac{1}{3}$ is written as 0.010101.... Since the computer stores only a finite number of digits, it cannot store $\frac{1}{3}$ accurately—it stores only an approximation.

A more distressing example is $\frac{1}{5}$. In the decimal system, it's .2, but in the binary system it's .0011001100110011.... So the computer can't handle $\frac{1}{5}$ accurately, even though a human can.

Suppose you run this BASIC program on a PDP-10 computer:

```
10 PRINT "MY FAVORITE NUMBER IS"4.001-4
20 END
```

The computer will try to convert 4.001 to binary. Unfortunately, it can't be converted exactly; the computer's binary approximation of it is slightly too small. The computer's final answer to 4.001-4 is therefore slightly less than the correct answer. Instead of printing MY FAVORITE NUMBER IS .001, the computer will print MY FAVORITE NUMBER IS .000999987.

If your computer isn't a PDP-10, it's approximation will be slightly different. To test your computer's accuracy, try 4.0001-4, and 4.00001-4, and 4.000001-4, etc. You might be surprised at its answers.

Let's see how the PDP-10 handles this:

```
10 FOR X = 7 TO 193 STEP .1
20     PRINT X
30 NEXT X
40 END
```

The computer will convert 7 and 193 to binary accurately, but will convert .1 to binary only approximately; the approximation is slightly too large. The last few numbers it should print are 192.8, 192.9, and 193, but because of the approximation it will print slightly more than 192.8, then slightly more than 192.9, and then stop (since it is not allowed to print anything over 193).

There are only two binary digits: 0 and 1. A *binary digit* is called a *bit*. For example, .001100110011 is a binary approximation of $\frac{1}{5}$ that consists of twelve bits. A sixteen-bit approximation of $\frac{1}{5}$ would be .0011001100110011. A bit that is 1 is called *turned on*; a bit that is 0 is *turned off*. For example, in the expression 11001, three bits are turned on and two are off. We also say that three of the bits are *set* and two are *cleared*.

All information inside the computer is coded, in the form of bits:

PART OF THE COMPUTER	WHAT A 1 BIT IS	WHAT A 0 BIT IS
electric wire	high voltage	low voltage
punched paper tape	a hole in the tape	no hole in the tape
punched IBM card	a hole in the card	no hole in the card
magnetic drum	a magnetized area	a non-magnetized area
core memory	a core magnetized clockwise	a core magnetized counterclockwis
flashing light	the light is on	the light is off

For example, to represent 11 on part of a punched paper tape, the computer punches two holes close together. To represent 1101, the computer punches two holes close together, and then another hole farther away.

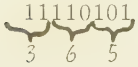
OCTAL

Octal is a short-hand notation for binary:

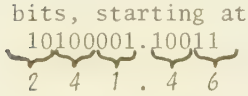
OCTAL	MEANING
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Each octal digit stands for three bits. For example, the octal number 72 is short for 111010.

To convert a binary integer to octal, divide the number into chunks of three bits, starting at the right. For example, here's how to convert 11110101 to octal:



To convert a binary real number to octal, divide the number into chunks of three bits, starting at the decimal point and working in both directions:



HEXADECIMAL

Hexadecimal is another short-hand notation for binary:

HEXADECIMAL	MEANING
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

For example, the hexadecimal number 4F is short for 01001111. To convert a binary number to hexadecimal, divide the number into chunks of 4 bits, starting at the decimal point and working in both directions:

```
 11010110100.1111111
  6  B  4 . F  E
```

CHARACTER CODES

ASCII

To store a character in a string, the computer uses a code. The most famous code is the *American Standard Code for Information Interchange (ASCII)*, which has 7 bits for each character. Here are examples:

CHARACTER	ASCII CODE	ASCII CODE IN OCTAL
space	0100000	040
!	0100001	041
"	0100010	042
#	0100011	043
$	0100100	044
%	0100101	045
&	0100110	046
'	0100111	047
(	0101000	050
)	0101001	051
*	0101010	052
+	0101011	053
,	0101100	054
-	0101101	055
.	0101110	056
/	0101111	057

CHARACTER	ASCII CODE	ASCII CODE IN OCTAL
0	0110000	060
1	0110001	061
2	0110010	062
etc.		
9	0111001	071
:	0111010	072
;	0111011	073
<	0111100	074
=	0111101	075
>	0111110	076
?	0111111	077
@	1000000	100
A	1000001	101
B	1000010	102
C	1000011	103
etc.		
Z	1011010	132
[	1011011	133
\	1011100	134
]	1011101	135
↑	1011110	136
←	1011111	137

"ASCII" is pronounced "ass key".

PDP-10 computers use mainly 7-bit ASCII, but can also handle a "6-bit ASCII", formed by eliminating ASCII's second bit: for example, the 6-bit ASCII code for the symbol $ is 0 00100. PDP-8 computers use mainly a "6-bit ASCII" formed by eliminating 7-bit ASCII's leftmost bit, but can also handle an "8-bit ASCII" formed by putting a 1 before 7-bit ASCII. The PDP-11 and most microcomputers use an "8-bit ASCII" formed by putting a 0 before 7-bit ASCII. Most terminals use 7-bit ASCII. CDC computers use a special CDC 6-bit code.

EBCDIC

Instead of using ASCII, IBM computers use the *Extended Binary-Coded-Decimal Interchange Code (EBCDIC)*, which has 8 bits for each character. Here are examples:

CHARACTER	EBCDIC CODE IN HEXADECIMAL	CHARACTER	EBCDIC CODE IN HEXADECIMAL	CHARACTER	EBCDIC CODE IN HEXADECIMAL
space	40	-	60	A	C1
¢	4A	/	61	B	C2
<	4C	,	6B	etc.	
(	4D	%	6C	I	C9
+	4E	_	6D	J	D1
|	4F	>	6E	K	D2
&	50	?	6F	etc	
!	5A	:	7A	R	D9
$	5B	#	7B	S	E2
*	5C	@	7C	T	E3
)	5D	'	7D	etc.	
;	5E	=	7E	Z	E9
¬	5F	"	7F	0	F0
				1	F1
				etc.	
				9	F9

"EBCDIC" is pronounced "ebb sih Dick".

IBM 360 computers can also handle an "8-bit ASCII", formed by copying ASCII's first bit after the second bit: for example, the 8-bit ASCII code for the symbol $ is 01000100. But IBM 370 computers (which are newer than IBM 360 computers) don't bother with ASCII: they stick strictly with EBCDIC.

80-column IBM cards use *Hollerith code*, which resembles EBCDIC but has 12 bits instead of 8. 96-column IBM cards use a 6-bit code that's an abridgement of the Hollerith code.

Here's a program in BASIC:

```
10 IF "A"<"9" THEN 100
20 PRINT "CAT"
30 STOP
100 PRINT "DOG"
110 END
```

Which will the computer print: CAT or DOG? The answer depends on whether the computer uses EBCDIC or ASCII.

Suppose the computer uses EBCDIC. Then the code for "A" is hexadecimal C1, and the code for "9" is hexadecimal F9. Since C1 is less than F9, the computer considers "A" to be less than "9", so the computer prints DOG.

But if the computer uses 7-bit ASCII instead of EBCDIC, the code for "A" is octal 101, and the code for "9" is octal 071; since 101 is greater than 071, the computer considers "A" to be greater than "9", so the computer prints CAT.

BYTES

A *byte* usually means: eight bits. For example, here's a byte: 10001011.

For computers that use 7-bit ASCII, programmers sometimes define a byte to be 7 bits instead of 8. For computers that use 6-bit ASCII, programmers sometimes define a byte to be 6 bits. So if someone tries to sell you a computer whose memory can hold "16,000 bytes", he probably means 16,000 8-bit bytes, but might mean 7-bit bytes or 6-bit bytes.

NIBBLES

A *nibble* is 4 bits. It's half of an 8-bit byte. Since a hexadecimal digit stands for 4 bits, a hexadecimal digit stands for a nibble.

ASSEMBLY LANGUAGE

SEXY ASS

In this chapter, you'll learn the fundamental concepts of assembly language, quickly and easily.

Unfortunately, different CPU's have different assembly languages.

I've invented an assembly language that combines the best features of all the other assembly languages. My assembly language is called *SEXY ASS*, because it's a Simple EXcellent-for-You ASSembler.

After you study the mysteries of the SEXY ASS, you can easily get your rear in gear and become the dominant master of the assemblers sold for Apple, Radio Shack, IBM, DEC, etc. Mastering them will become so easy that you'll say, "Assembly language is a piece of cheesecake!"

To begin, let's get a close-up view of the SEXY ASS. . . .

ELECTRONIC BOXES

The computer's guts consists of two main parts: the brain (which is called the *CPU*) and the *main memory* (which consists of RAM and ROM).

Inside the CPU are many electronic boxes, called *registers*. Each register holds several electrical signals; each signal is called a *bit*; so each register holds several bits. Each bit is either 1 or 0. A "1" represents a high voltage; a "0" represent a low voltage. If the bit is 1, the bit is said to be *high* or *on* or *set* or *true*; if the bit is 0, the bit is said to be *low* or *off* or *cleared* or *false*.

The CPU's most important register is called the *accumulator* (or *A*). In the SEXY ASS system, the accumulator consists of 8 bits, which is 1 byte. (Later, I'll explain how to make the CPU handle several bytes simultaneously; but the accumulator itself holds only 1 byte.)

Like the CPU, the main memory consists of electronic boxes. The electronic boxes *in the CPU* are called *registers*, but the electronic boxes *in the main memory* are instead called *memory locations*. Because the main memory acts like a gigantic post office, the memory locations are also called *addresses*. In the SEXY ASS system, each memory location holds 1 byte. There are many *thousands* of memory locations; they're numbered 0, 1, 2, 3, etc.

When using SEXY ASS, you can type numbers in decimal, binary, or hexadecimal. (For SEXY ASS, octal isn't useful.) For example, the number "twelve" is written "12" in decimal, "1100" in binary, and "C" in hexadecimal. To indicate which number system you're using, put a percent sign in front of each binary number, and put a dollar sign in front of each hexadecimal number. For example, in SEXY ASS you can write the number "twelve" as either 12 or %1100 or $C. (In that respect, SEXY ASS copies the 6502 assembly language, which also uses the percent sign and the dollar sign.)

Most of the time, we'll be using hexadecimal, so let's quickly review what hexadecimal is all about. To count in hexadecimal, just start counting as you learned in elementary school ($1, $2, $3, $4, $5, $6, $7, $8, $9); but after $9, you continue counting by using the letters of the alphabet ($A, $B, $C, $D, $E, and $F). After $F (which is fifteen), you say $10 (which means sixteen), then say $11 (which means seventeen), then $12, then $13, then $14, etc., until you reach $19; then come $1A, $1B, $1C, $1D, $1E, and $1F. Then come $20, $21, $22, etc., up to $29, then $2A, $2B, $2C, $2D, $2E, and $2F. Then comes $30. Eventually, you get up to $99, then $9A, $9B, $9C, $9D, $9E, and $9F. The come $A0, $A1, $A2, etc., up to $AF. Then come $B0, $B1, $B2, etc., up to $BF. You continue that pattern, until you reach $FF. Get together with your friends, and try counting up to $FF. (Don't bother pronouncing the dollar signs.) Yes, you too can count like a pro!

Each hexadecimal digit represents 4 bits. Therefore, an 8-bit byte requires *two* hexadecimal digits. So a byte can be anything from $00 to $FF.

I said that the main memory consists of *thousands* of memory locations, numbered 0, 1, 2, etc. The most important part of the main memory consists of 65,536 memory locations (64K), which are numbered from 0 to 65,535. Programmers usually number them in hexadecimal: the hexadecimal numbers go from $0000 to $FFFF. ($FFFF in hexadecimal is the same as 65,535 in decimal.) Later, I'll explain how to handle beyond 64K; but for now, let's restrict our attention to just 64K.

HOW TO COPY A BYTE

Here's a simple, one-line program, written in the SEXY ASS assembly language:

```
LOAD    $7000
```

It makes the computer copy one byte, from memory location $7000 to the accumulator. So after the computer obeys that instruction, the accumulator will contain the same data as the memory location. For example, if the memory location contains the byte %01001111 (which can also be written as $4F), so will the accumulator.

Notice the spaces before and after the word LOAD. To make those spaces, hit the TAB key (or press the space bar many times).

The word LOAD tells the computer to copy from a memory location to the accumulator. The opposite of the word LOAD is the word STORE: it tells the computer to copy from the accumulator to a memory location. For example, if you type—

```
STORE   $7000
```

the computer will copy a byte from the accumulator to memory location $7000.

Problem: write an assembly-language program that copies a byte from memory location $7000 to memory location $7001. Solution: you must do it in two steps. First, copy from memory location $7000 to the accumulator (by using the word LOAD); then copy from the accumulator to memory location $7001 (by using the word STORE). Here's the program:

```
LOAD    $7000
STORE   $7001
```

ARITHMETIC

If you say—

```
INC
```

the computer will increment (increase) the number in the accumulator, by adding 1 to it. For example, if the accumulator contains the number $25, and you then say INC, the accumulator will contain the number $26. For another example, if the accumulator contains the number $39, and you then say INC, the accumulator will contain the number $3A (because, in hexadecimal, after 9 comes A).

Problem: write a program that increments the number that's in location $7000; for example, if location $7000 contains $25, the program should change that data, so that location $7000 contains $26 instead. Solution: copy the number from location $7000 to the accumulator, then increment the number, then copy it back to location $7000. . . .

```
LOAD    $7000
INC
STORE   $7000
```

That example illustrates the fundamental rule of assembly-language programming, which is: to manipulate a memory location's data, copy the data to the accumulator, manipulate the accumulator, and then copy the revised data from the accumulator to memory.

The opposite of INC is DEC: it *decrements* (decreases) the number in the accumulator, by subtracting 1 from it.

If you say—

```
ADD     $7000
```

the computer will change the number in the accumulator, by adding to it the number that was in memory location $7000. For example, if the accumulator had contained the number $16, and memory location $7000 had contained the number $43, the number in the accumulator will change and become the sum, $59. The number in memory location $7000 will remain unchanged: it will still be $16.

Problem: find the sum of the numbers in memory locations $7000, $7001, and $7002, and put that sum into memory location $7003. Solution: copy the number from memory location $7000 to the accumulator, then add to the accumulator the numbers from memory locations $7001 and $7002, so that the accumulator contains the sum of all three numbers; finally, copy the sum from the accumulator to memory location $7003. . . .

```
LOAD    $7000
ADD     $7001
ADD     $7002
STORE   $7003
```

The opposite of ADD is SUB, which means SUBtract. If you say SUB $7000, the computer will change the number in the accumulator, by subtracting from it the number in memory location $7000.

IMMEDIATE ADDRESSING

If you say—

```
LOAD    #$25
```

the computer will put the number $25 into the accumulator. The $25 is the data. In the instruction "LOAD #$25", the symbol "#" tells the computer that the $25 is the data instead of being a memory location.

If you were to omit the #, the computer would assume the $25 meant memory location $0025, and so the computer would copy data from memory location $0025 to the accumulator.

An instruction that contains the symbol # is said to be an *immediate* instruction; it is said to use *immediate* addressing. Such instructions are unusual.

The more usual kind of instruction, which does *not* use the symbol #, is called a *direct* instruction.

Problem: change the number in the accumulator, by adding $12 to it. Solution:

```
ADD     #$12
```

Problem: change the number in memory location $7000, by adding $12 to that number. Solution: copy the number from memory location $7000 to the accumulator, add $12 to it, and then copy the sum back to the memory location. . . .

```
LOAD    $7000
ADD     #$12
STORE   $7000
```

Problem: make the computer find the sum of $16 and $43, and put the sum into memory location $7000. Solution: put $16 into the accumulator, add $43 to it, and then copy the sum from the accumulator to memory location $7000. . . .

```
LOAD    #$16
ADD     #$43
STORE   $7000
```

VIDEO RAM

The *video RAM* is part of the computer's RAM, and holds a copy of what's on the screen.

For example, suppose you're running a program that analyzes taxicabs, and the screen (of your TV or monitor) shows information about various cabs. If the upper-left corner of the screen shows the word CAB, the video RAM contains the ASCII code numbers for the letters C, A, and B. Since the ASCII code number for C is 67 (which is $43), and the ASCII code number for A is 65 (which is $41), and the ASCII code number for B is 66 (which is $42), the video RAM contains $43, $41, and $42. The $43, $41, and $42 represent the word CAB.

Suppose that the video RAM begins at memory location $6000. If the screen's upper-left corner shows the word CAB, memory location $6000 contains the code for C (which is $43); the next memory location ($6001) contains the code for A (which is $41); and the next memory location ($6002) contains the code for B (which is $42).

Problem: assuming that the video RAM begins at location $6000, make the computer write the word CAB onto the screen's upper left corner. Solution: write $43 into memory location $6000, write $41 into memory location $6001, and write $42 into memory location $6002. . . .

```
LOAD    #$43
STORE   $6000
LOAD    #$41
STORE   $6001
LOAD    #$42
STORE   $6002
```

The computer knows that $43 is the code number for "C". When you're writing that program, if you're too lazy to figure out the $43, you can simply write "C"; the computer will understand. So you can write the program like this:

```
LOAD    #"C"
STORE   $6000
LOAD    #"A"
STORE   $6001
LOAD    #"B"
STORE   $6002
```

That's the solution if the video RAM begins at memory location $6000. On *your* computer, the video RAM might begin at a different memory location instead. To find out about *your* computer's video RAM, look at the back of the technical manual that came with your computer. There you'll find a *memory map*: it shows which memory locations are used by the video RAM, which memory locations are used by other RAM, and which memory locations are used by the ROM.

LABELS

You can give each line of your program a name. For example, you can give a line the name FRED. To do so, put the name FRED at the beginning of the line, like this:

```
FRED    LOAD    $7000
```

The line's name (FRED) is at the left margin. The command itself (LOAD $7000) is indented, by pressing the TAB key. In that line, FRED is called the *label*, LOAD is called the *operation* or *mnemonic*, and $7000 is called the *address*.

Languages such as BASIC let you say "GO TO". In assembly language, you say "JUMP" instead of "GO TO". For example, to make the computer GO TO the line named FRED, say:

```
JUMP    FRED
```

The computer will obey: it will JUMP to the line named FRED.

I've been explaining assembly language. *Machine language* resembles assembly language; what's the difference?

To find out, let's look at a machine language called *SEXY MACHO* (because it's a Simple EXcellent-for-You MACHine language Original).

SEXY MACHO resembles SEXY ASS; here are the major differences. . . .

In SEXY ASS assembly language, you use words such as LOAD, STORE, INC, DEC, ADD, SUB, and JUMP. Those words are called *operations* or *mnemonics*. In SEXY MACHO machine language, you replace those words by code numbers: the code number for LOAD is 1; the code number for STORE is 2; INC is 3; DEC is 4; ADD is 5; SUB is 6; and JUMP is 7. The code numbers are called the *operation codes* or *op codes*.

In SEXY ASS assembly language, the symbol "#" indicates immediate addressing; a lack of the symbol "#" indicates direct addressing instead. In SEXY MACHO machine language, you replace the symbol "#" by the code number 1; if you want direct addressing instead, you must use the code number 0.

In SEXY MACHO, all code numbers are hexadecimal.

For example, look at this SEXY ASS instruction:

```
ADD     #$43
```

To translate that instruction into SEXY MACHO machine language, just replace each symbol by its code number. Since the code number for ADD is 5, and the code number for # is 1, the SEXY MACHO version of that line is:

```
5143
```

Let's translate STORE $7003 into SEXY MACHO machine language. Since the code for STORE is 2, and the code for direct addressing is 0, the SEXY MACHO version of that command is:

```
207003
```

In machine language, you can't use any words or symbols: you must use their code numbers instead. To translate a program from assembly language to machine language, you must look up the code number of each word or symbol.

An *assembler* is a program that makes the computer translate from assembly language to machine language.

The CPU understands only machine language: it understands only numbers. It does *not* understand assembly language: it does not understands words and symbols. If you write a program in assembly language, you must buy an assembler, which translates your program from assembly language to machine language, so that the computer can understand it.

Since assembly language uses English words (such as LOAD), assembly language seems more "human" than machine language (which uses code numbers). Since programmers are humans, programmers prefer assembly language over machine language. Therefore, the typical programmer writes in assembly language, and then uses an assembler to translate the program to machine language, which is the language that the CPU ultimately requires.

Here's how the typical assembly-language programmer works. First, the programmer types the assembly-language program, and uses a word processor to help edit it. The word processor automatically puts the assembly-language program onto a disk. Next, the programmer uses the assembler, to translate the assembly-language program into machine language. The assembler puts the machine-language version of the program onto the disk. So now the disk contains *two* versions of the program: the disk contains the original version (in assembly language) and also contains the translated version (in machine language). The original version (in assembly language) is called the *source code*; the translated version (in machine language) is called the *object code*. Finally, the programmer gives a command that makes the computer copy the machine-language version (the object code) from the disk to the RAM and run it.

Here's a tough question: how does the assembler translate "JUMP FRED" into machine language? Here's the answer. . . .

The assembler realizes that FRED is the name for a line in your program. The assembler hunts through your program, to find out which line is labeled FRED. When the assembler finds that line, it analyzes that line, to figure out where that line will be in the RAM, after the program is translated into machine language and running. For example, suppose the line that's labeled FRED will become a machine-language line which, when the program is running, will be in the RAM at memory location $2053. Then "JUMP FRED" must be translated into this command: "jump to the machine-language line that's in the RAM at memory location $2053". So "JUMP FRED" really means:

```
JUMP    $2053
```

Since the code number for JUMP is 7, and the addressing isn't immediate (and therefore has code 0 instead of 1), the machine language version of JUMP FRED is:

```
702053
```

SYSTEM SOFTWARE

The computer's main memory consists of RAM and ROM. In a typical computer, the first few memory locations ($0000, $0001, $0002, etc.) are ROM: they permanently contain a program called the *bootstrap*, which is written in machine-language.

When you turn on the computer's power switch, the computer automatically runs the bootstrap program. If your computer uses disks, the bootstrap program makes the computer start reading information from the disk in the main drive. In fact, it makes the computer copy a machine language program from the disk to the RAM. The machine-language program that it copies is called the *DOS*.

After the DOS has been copied to the RAM, the computer starts running the DOS program. The DOS program makes the computer print a message on the screen (such as "Welcome to CP/M" or "Welcome to MS-DOS") and print a symbol on the screen (such as "A>") and then wait for you to type a command.

That whole procedure is called *bootstrapping* (or *booting up*), because of the phrase "pull yourself up by your own bootstraps". By using the bootstrap program, the computer pulls itself up to new intellectual heights: it becomes a CP/M machine or an MS-DOS machine or an Apple DOS machine or a TRSDOS machine.

After booting up, you can start writing programs in BASIC. But how does the computer understand the BASIC words, such as PRINT, INPUT, IF, THEN, and GO TO? Here's how. . . .

While you're using BASIC, the computer is running a machine-language program, that makes the computer *seem* to understand BASIC. That machine-language program, which is in the computer's ROM or RAM, is called the *BASIC language processor* or *BASIC interpreter*. If your computer uses *Microsoft* BASIC, the BASIC interpreter is a machine-language program that was written by Microsoft Incorporated (a "corporation" that consists of Bill Gates and his pals).

FLAGS

CARRY FLAG

A byte consists of 8 bits. The smallest number you can put into a byte is %00000000. The largest number you can put into a byte is %11111111, which in hexadecimal is $FF, and which in decimal is 255.

What happens if you try to go higher than %11111111? To find out, examine this program:

```
LOAD    #%10000001
ADD     #%10000010
```

In that program, the top line puts the binary number %10000001 into the accumulator. The next line tries to add %10000010 to the accumulator. But the sum, which is %100000011, contains 9 bits instead of 8, and therefore can't fit into the accumulator.

The computer splits that sum into two parts: the left bit (1) and the remaining bits (00000011). The left bit (1) is called the *carry bit*; the remaining bits (00000011) are called the *tail*. Since the tail contains 8 bits, it fits nicely into the accumulator; so the computer puts it into the accumulator. The carry bit is put into a special place inside the CPU; that special place is called the *carry flag*.

So that program makes the accumulator become 00000011, and makes the carry flag become 1.

Here's an easier program:

```
LOAD    #%1
ADD     #%10
```

The top line puts %1 into the accumulator; so the accumulator's 8 bits are %00000001. The bottom line adds %10 to the number in the accumulator; so the accumulator's 8 bits become %00000011. Since the numbers involved in that addition were so small, there was no need for a 9th bit—no need for a carry bit. To emphasize that no carry bit was required, the carry flag automatically becomes 0.

Here's the rule: if an arithmetic operation (such as ADD, SUB, INC, or DEC) gives a result that's too long to fit into 8 bits, the carry flag becomes 1; otherwise, the carry flag becomes 0.

NEGATIVES

The largest number you can fit into a byte is %11111111, which in decimal is 255. Suppose you try to add 1 to it. The sum is %100000000, which in decimal is 256. But since %100000000 contains 9 bits, it's too long to fit into a byte. So the computer sends the leftmost bit (the 1) to the carry flag, and puts the tail (the 00000000) into the accumulator. As a result, the accumulator contains 0.

So in assembly language, if you tell the computer to do %11111111+1 (which is 255+1), the accumulator says the answer is 0 (instead of 256).

So in assembly language, %11111111+1 is 0. In other words, %11111111 solves the equation x+1=0.

According to high-school algebra, the equation x+1=0 has this solution: x=-1. But we've seen that in the assembly language, the equation x+1=0 has the solution x=%11111111. Conclusion: in assembly language, -1 is the same as %11111111.

Now you now that -1 is the same as %11111111, which is 255. Yes, -1 is the same as 255. Similarly, -2 is the same as 254; -3 is the same as 253; -4 is the same as 252. Here's the general formula: -n is the same as 256-n. (That's because 256 is the same as 0.)

%11111111 is 255 and is also -1. Since -1 is a shorter name than 255, we say that %11111111 is *interpreted as* -1. Similarly, %11111110 is 254 and also -2; since -2 is a shorter name than 254, we say that %11111110 is interpreted as -2. At the other extreme, %00000010 is 2 and is also -254; since 2 is a shorter name than -254, we say that %11111110 is interpreted as 2. Here's the rule: if a number is "almost" 256, it's interpreted as a negative number; otherwise, it's interpreted as a positive number.

How high must a number be, in order to be "almost" 256, and therefore to be interpreted as a negative number? The answer is: if the number is at least 128, it's interpreted as a negative number. Putting it another way, if the number's leftmost bit is 1, it's interpreted as a negative number.

That strange train of reasoning leads to the following definition: a *negative number* is a byte whose leftmost bit is 1.

A byte's leftmost bit is therefore called the *negative bit* or the *sign bit*.

FLAG REGISTER

You've seen that the CPU contains a register called the *accumulator*. The CPU also contains a second register, called the *flag register*. In the SEXY ASS system, the flag register contains 8 bits (one byte). Each of the 8 bits in the flag register is called a *flag*; so the flag register contains 8 flags.

Each flag is a bit: it's either 1 or 0. If the flag is 1, the flag is said to be *raised* or *set*. If the flag is 0, the flag is said to be *lowered* or *cleared*.

One of the 8 flags is the carry flag: it's raised (becomes 1) whenever an arithmetic operation requires a 9th bit. (It's lowered whenever an arithmetic operation does *not* require a 9th bit.)

Another one of the flags is the *negative flag*: it's raised whenever the number in the accumulator becomes negative. For example, if the accumulator becomes %11111110 (which is -2), the negative flag is raised (i.e. the negative flag becomes 1). It's lowered whenever the number in the accumulator becomes *non*-negative.

Another one of the flags is the *zero flag*: it's raised whenever the number in the accumulator becomes zero. (It's lowered whenever the number in the accumulator becomes *non*-zero.)

You can say—

```
        JUMPN   FRED
```

That means: JUMP to the line whose name is FRED, if the Negative flag is raised. So the computer will JUMP to FRED if a negative number was recently put into the accumulator. (If a *non*-negative number was recently put into the accumulator, the computer will *not* jump to FRED.)

JUMPN means "JUMP if the Negative flag is raised." JUMPC means "JUMP if the Carry flag is raised." JUMPZ means "JUMP if the Zero flag is raised."

JUMPNL means "JUMP if the Negative flag is Lowered." JUMPCL means "JUMP if the Carry flag is Lowered." JUMPZL means "JUMP if the Zero flag is Lowered."

Problem: make the computer look at memory location $7000; if the number in that memory location is negative, make the computer jump to a line named FRED. Solution: copy the number from memory location $7000 to the accumulator, to influence the Negative flag; then JUMP if Negative. . . .

```
        LOAD    $7000
        JUMPN   FRED
```

Problem: make the computer look at memory location $7000. If the number in that memory location is negative, make the computer print a minus sign in the upper-left corner of the screen; if the number is positive instead, make the computer print a plus sign instead; if the number is zero, make the computer print a zero. Solution: copy the number from memory location $7000 to the accumulator (by saying LOAD); then analyze that number (by using JUMPN and JUMPZ); then LOAD the ASCII code number for either "+" or "-" or "0" into the accumulator (whichever is appropriate); finally, copy that ASCII code number from the accumulator to the video RAM (by saying STORE). . . .

```
        LOAD    $7000
        JUMPN   NEGAT
        JUMPZ   ZERO

        LOAD    #"+"
        JUMP    DISPLAY

NEGAT   LOAD    #"-"
        JUMP    DISPLAY

ZERO    LOAD    #"0"

DISPLAY STORE   $6000
```

SCHEDULING THE USERS

CPU

The first computers were invented about 1950. Back then, the only kind of computer you could buy was a *single user system*: only one person could use the computer. If two people wanted to use the computer, the second person had to stand in line behind the first person, until the first person finished.

Batch

The first improvement to that system was *batch*. In a computer that had *batch processing*, the second person didn't have to stand in line to use the computer. Instead, the second person could feed his program into the computer also, even though the first person wasn't done. The second person's program waited in a special part of the computer's memory (RAM or core or disk or drum or tape), until the first person's program was done; then the computer would start working on the second program.

It was called a *batch* system, because the people could insert a batch of two programs, instead of inserting just one program at a time. In most batch systems, the people could insert a batch of *many* programs: the computer would do each one in order.

Multiprogramming

A computer system consists of many parts: the CPU, the printer, the disk (or tape), etc. Some parts of the system work more slowly than others.

For example, the printer is slower than the CPU. If the CPU tells the printer to print some information, the CPU must wait until the printer finishes.

While the CPU is waiting, why not let the CPU start working on the second person's program? If the CPU is smart enough to do that, it's called a *multiprogramming system*.

So in a simple multiprogramming system, the CPU uses this strategy: the CPU begins by working on the first person's program; but if work on that program must be temporarily interrupted (because of waiting for the printer, or some other slow device), the CPU starts working on the second program. (Similarly, if the second program must be temporarily interrupted also, the CPU starts working on a third program.) But the first program always has top priority: as soon as the first program can continue (because the printer has finished obeying the CPU's command), work on the first program resumes, and all the other programs are put on hold again.

A difficulty

That simple multiprogramming system is quite practical and efficient— except for the following difficulty....

Suppose two guys walk up to the computer. One guy's program is very long (it figures out the entire payroll for a large company); the other guy's program is short (it just adds 2+2). If the guy with the longer program is kind, he'll let the other guy go first. But what if the guy with the long program is mean and shoves his way to the computer first? Then the poor second guy (who wanted to just add 2+2) will have to wait until the entire payroll finishes (unless the first program uses the printer, which makes the CPU switch attention to the second program).

It would be nice if the CPU could "psyche out" the situation, and give attention to the second guy, without waiting for the first guy to finish. That way, the second guy could leave in a few seconds, instead of having to wait many hours.

Time-slicing

The easiest way to make the CPU get to the second guy quickly is to use a technique called *round-robin time-slicing*. On most large computers, the "time-slice" is a sixtieth of a second. Here's how it works....

The CPU works on the first guy's program for a sixtieth of a second. Then the CPU switches its attention to the second guy, and gives him a sixtieth of a second. Then the CPU switches to the third guy, and gives him a sixtieth of a second. And so on, until everybody's program has received a sixtieth of a second. Then the CPU returns to the first program, and gives it another sixtieth of a second. Then— to the second program. And so on. If somebody's program is short, he'll be done fairly soon, and won't have to wait several hours for the long-winded guys ahead of him to finish.

If the computer uses that round-robin time-slicing, each person has his own terminal. The many terminals are attached to the computer. The CPU goes round and round, switching its attention from terminal to terminal every sixtieth of a second.

It all happens so fast that each person is under the illusion that the CPU is thinking about him the whole time. But really, the CPU is paying attention to just one person at a time, and switches its attention from person to person very fast. That's called *timesharing*. It's where the CPU juggles several programs so quickly that each person is under the *illusion* that the CPU is thinking about just him the whole time.

In that system, a person might not get a full sixtieth of a second on his turn. For example, if the person's program needs to use the printer, the CPU sends some data out to the printer but then immediately moves on to the next person, without waiting for the printer to catch up. When the CPU has given all the other people their time slices and returns again to the person who wanted the printer, the CPU checks whether the printer has finished the job yet.

Fewer switches

Unfortunately, "switching attention from one person to another" is complicated: the CPU needs several milliseconds to make the switch. It would be nice if the CPU could switch less often, yet still get to each person fast enough so that each person still thinks the CPU is thinking only about him.

For example, suppose a person is a "CPU hog": he's running a program that won't finish for several hours. Instead of giving that person many little time-slices, the CPU could act more efficiently by by totally ignoring that person for several hours (which will make everybody else in the computer room quite happy!), and then give that person a solid block of time toward the end of those hours. He'll never know the difference: his job will finish at the same time as it would have otherwise. And the CPU will have an easier time, since it doesn't have to switch its attention to and from him so many times.

But how can the CPU decide which person is the "hog"? Make the CPU count how many time-slices and how much RAM each person has already been using; if a person's count is high, the person has been acting hoggish. The CPU assumes that the person will probably continue to be a hog; so that person is ignored until later, when he's given a solid block of time.

But if the "solid block of time" is *too* long, all the other people will be ignored for too long and will think the computer broke. So the "solid block of time" shouldn't be more than a few seconds. Nevertheless, "a few seconds" is a lot longer than a *sixtieth* of a second! If the program still isn't done after the solid block of time, give the program *another* solid block of time, later.

The PDP-20 and other top-notch timesharing systems use those strategies.

Contrasts

Remember: a computer that has "batch" might not be fancy enough to have "multiprogramming"; and a computer that has "multiprogramming" might not be fancy enough to have "timesharing".

During the 1970's, a popular question was, "Does your computer use timesharing, or just batch?"

Most timesharing systems use terminals with screens; most just batch systems use punched cards instead. Timesharing (using terminals that have screens) is becoming more popular than just batch (and just punched card).

Distributed intelligence

The whole point of that discussion was to show how to make many people share a single CPU efficiently. But why bother? The CPU chip inside a Radio Shack or Apple computer costs less than $10; why bother sharing it? Why not simply give each person his own CPU?

So today many people are getting rid of maxicomputers that have fancy timesharing operating systems, and are replacing them by a collection of microcomputers, each of which handles only one person at a time.

That's called *distributed intelligence*: tying together many little CPU's, instead of forcing everybody to share one big CPU.

MEMORY

Suppose you want to run a program that's too long to fit in the RAM. What should you do? The obvious answer: chop the program into little pieces, and run one piece at a time.

Virtual memory

If you buy a computer that has *virtual memory*, the computer will chop the program automatically. The operating system will automatically chop your program into little pieces, put as many pieces as possible into the RAM, and run them. The remaining pieces are put onto a disk instead. When the CPU finishes processing the pieces in the RAM, and needs to get to the pieces on the disk, the operating system automatically fetches those pieces from the disk and copies them into the RAM— after copying the no-longer-needed pieces from the RAM to the disk.

Each piece of the program is called a *page*. On most computers, each page is 4K. Copying a page from the disk to the RAM (or vice versa) is called *paging* or *swapping*.

If many people try to share the RAM, the operating system tries to fit all their programs into the RAM simultaneously. But if they won't all fit, the operating system automatically puts some of their programs onto the disk temporarily, until the other programs are done (or have used up a time-slice).

Thrashing

If too many people try to use the computer at once, and the computer's RAM is too small, the operating system will spend lots of time moving programs (or pages) to and from the disk. That's bad. In that situation, we say the operating system is *thrashing*. It's spending so much time moving between the RAM and the disk that hardly any time remains for the CPU to run the programs! The computer system is spending more time on bureaucracy than on doing useful work! The solution: buy more RAM, or tell some of the people to go home!

I/O DEVICES

I/O devices (such as printers and terminals) are much slower than the CPU. After the CPU sends a command to a slow device, the CPU must wait for the device to obey. While the CPU is waiting, the CPU ought to be doing something useful— such as processing somebody else's program.

Buffers

Here's how the CPU handles the problem of "waiting". When the CPU wants to send a message (or command) to a slow device, it puts the message into a *buffer*, which is a special part of the RAM. The device peeks at the buffer, and obeys what the message says. While the device is handling the message in the buffer, the CPU's attention switches to a different program— or to a different part of the same program.

The buffer is in RAM. But *which* RAM? If you buy a new computer system, each I/O device contains a little RAM (usually 2K). The "little RAM inside the I/O device" is the buffer. That's if your equipment is ultra-modern. If your equipment is old-fashioned, the buffer is instead part of the ordinary RAM (the RAM that's in the same cabinet as the CPU).

Buffers are used not only for printers and terminals but also for disk drives and tape drives. Each slow device needs its own buffer.

Spooling

Here's another nasty problem. Suppose your program says to write answers on the printer, but somebody else has been using the printer and hasn't finished yet.

In that situation, the CPU *pretends* to obey your command; it *pretends* to write your answers on the printer. But actually, it writes your answers on a disk instead. Then, when the previous person finishes using the printer, the CPU copies your answers from the disk to the printer (and erases them from the disk). That all happens automatically, thanks again to the operating system. The part of the operating system that takes care of waiting-for-the-printer is called the *spooler*. Putting your answers temporarily onto a disk is called *spooling*.

Spooling is used mainly for the printer, but can also be used for other output devices, such as the plotter.

The word "spool" is an abbreviation for "simultaneous processing of on-line" devices. If several people what to use the printer simultaneously, the operating system must know how to spool.

If you want to use the printer, and somebody else is in the middle of using the printer to print paychecks, and your operating system is so crummy that it doesn't have a spooler, what happens? Either the operating system tells you "Sorry, but I can't continue your program until the other guy is done— please wait" or the operating system makes a mess— by printing your answers in the middle of the other guy's paycheck!

Print queue

Suppose your operating system *does* have a spooler, so that it prints your answers temporarily onto a disk. What happens if a third person wants to use the printer also? Yep, that person's answers get printed onto the disk also. If the printer is very slow, many people might have their answers waiting on the disk. The answers that are waiting on the disk are called the *printer's waiting-line* or the *printer's queue* or (even more briefly) the *print queue*.

"Queue" is pronounced "kyoo"; it's a British word that means "waiting-line". In British supermarkets, the customers don't wait in lines, they "wait in queues"; they "queue up". So do the answers that are waiting to go to the printer.

PERSONALITY

COMPUTERS HAVE PERSONALITIES

A computer can be a boss, a conversationalist, or a drudge.

BOSS

A "boss" is a computer that monitors another machine. The "other machine" might be part of a factory's assembly line, a microwave oven, a burglar-alarm system, a series of traffic lights, the sound system for an electronic organ, or a life-support system that periodically gives intravenous injections to an unconscious patient. The computer makes sure everything is running smoothly. Usually, you won't even know the computer is there: it may be hidden inside the machine it monitors.

Instead of calling such a computer a "boss", computer ads call it a *controller* or a *scientific real-time system*.

CONVERSATIONALIST

A "conversationalist" is a computer you can chat with, by using a terminal or a keyboard-with-display. The conversation might be about a game you're playing against the computer, a reservation for an airplane flight, or your personality (if the computer is trying to play therapist). Throughout the conversation, what you say to the computer is brief, and so are the computer's replies. Typically, either you're asking the computer for more output, or the computer is asking you for more input.

Instead of calling such a computer a "conversationalist", computer ads call it an *interactive computer*. A conversationalist whose memory contains lots of data, and that answers questions about the data, is called a *data-inquiry system* or a *commercial real-time system*. A *timesharing computer* is a conversationalist that handles many programs simultaneously, with each program on a different terminal.

DRUDGE

A "drudge" is a computer that takes in piles of data and spits out piles of answers. Typically, the data is on cards, and the answers are on line-printer paper. While the computer is working, it doesn't ask you any questions; you never need to speak with it. It just does its job faithfully, blue-collar style.

A *batch processor* is a drudge that lets you start feeding in a second program before the first program has finished. If the batch processor starts *working on* the second program before the first program has finished— in other words, if it works on both programs simultaneously— it's called a *multiprogramming batch processor*.

OPERATING SYSTEMS

IBM

For the IBM 360 computer, IBM invented an Operating System called *OS*. IBM also invented a very stripped-down version of the operating system, called *Basic OS (BOS)*. If your computer had too little memory for OS, you'd use BOS instead. IBM also invented two intermediate versions, smaller than OS but larger than BOS. The intermediate versions were called the *Tape OS (TOS)* and the *Disk OS (DOS)*.

Nobody uses BOS or TOS anymore. So the only popular operating systems today are OS and DOS. If your computer has lots of memory, it uses OS; if it has too little memory for OS, it uses DOS instead.

To begin using an IBM computer, you must give commands that the operating system understands. The commands it understands are called the *Job Control Language (JCL)*. So before using an IBM computer, you must read a book that explains JCL. The version of JCL depends on which operating system you have: the JCL for OS is fancier than the JCL for DOS.

Your first JCL command tells the computer your name and account number and password, so the computer can decide whether you're an authorized user. Later, you must give a JCL command that tells the computer which language you want to use: BASIC, FORTRAN, PASCAL, COBOL, or a language that's weirder.

The original version of DOS was called simply "DOS". Later, IBM invented a better version of DOS, called *DOS with Virtual Storage (DOS/VS)*. To use DOS/VS, you must buy IBM's newer computer, which is called the IBM 370.

The original version of OS was called *OS with Multiprogramming a Fixed number of Tasks (OS/MFT)*. Then IBM invented a fancier version, called *OS with Multiprogramming a Variable number of Tasks (OS/MVT)*. Those versions of OS work on the IBM 360. For IBM's newer computer— the IBM 370— IBM invented *OS/VS1*, then an even fancier version, called *OS/VS2*.

For flexibility, get the *Virtual Machine operating system (VM)*, which is an operating system that lets you invent your own operating system, and lets you run five different operating systems simultaneously! If you use VM, you'll probably make one of your five operating systems be OS/VS1 (or OS/VS2), and make another one be *CMS* (which stands for *Cambridge Monitor System* or *Conversational Monitor System*).

DEC

DEC's largest computer is called the PDP-20 or Decsystem-20. Its operating system is called *The OPerating System 20 (TOPS-20)*. DEC's previous large computer was called the PDP-10 or Decsystem-10; its operating system was called *TOPS-10*. The commands that the operating system understands are called *monitor commands*. So if your computer is a PDP-20 or PDP-10, you must learn DEC's monitor commands (instead of learning IBM's JCL).

DEC also makes a popular minicomputer, called the PDP-11. Several operating systems are available for the PDP-11. The most popular operating system is called the *Resource-Sharing Timesharing System Extended (RSTS/E)*. It can handle many terminals, and is intended for programming in BASIC-PLUS or BASIC-PLUS-2. If your PDP-11 is too small to handle RSTS/E, get DEC's smaller operating system, which is called *Real-Time 11 (RT-11)*; it handles fewer terminals and is intended for FORTRAN or a stripped-down version of BASIC (called BASIC-11, which is worse than BASIC-PLUS or BASIC-PLUS-2). If your PDP-11 is large, and you want it to handle a large program in COBOL (instead of many small programs in BASIC), get the *Resource Sharing eXtended 11 operating system (RSX-11)* or a souped-up version of it, called the *Interactive Application System (IAS)*. If you're programming in a weird language called MUMPS, get the *Data System Manager 11 (DSM-11)* instead.

DEC's newest computer is a very fancy minicomputer (fancier than the PDP-11) and is called the Virtual Address eXtended 11 (VAX-11). Its operating system is called the *VAX Virtual Memory System (VAX/VMS)*. To give a command to VAX/VMS, you use *DEC Control Language (DCL)*— instead of IBM's JCL.

APPLE

For many years, Apple's disk operating system was called *Apple DOS version 3.2*. It organized the disk into 13 sectors.

Recently, Apple Computer Company has invented a better version, called *Apple DOS version 3.3*. It organizes the disk into 16 sectors. It lets you put more information onto the disk than version 3.2.

If you buy an Apple today, the disk controller assumes the disk has 16 sectors. If you try to feed your new Apple (16 sectors) an old disk (13 sectors), your Apple will vomit the disk, unless you feed the Apple beforehand a program called MUFFIN or a program called BASICS, which act like Alka-Seltzer.

CP/M

Many microcomputers that have disks use an operating system called *Control Program for Microcomputers (CP/M)*. The fanciest business programs from microcomputers assume you have CP/M. To get CP/M, you buy a disk that contains it. The disk costs about $150. The disk works only if your computer contains a Z-80 CPU (or its predecessor, the 8080 CPU). Most microcomputers contain a Z-80 CPU, but the Apple does not: the Apple contains a 6502 CPU instead. To use CP/M on an Apple, you must put a Z-80 CPU inside your Apple, by using Microsoft's Z-80 Softcard, which is a card that contains a Z-80 CPU and fits in your Apple.

CP/M was invented by a company called Digital Research, which has invented an even fancier version, called *Multi Programming for Microcomputers (MP/M)*. MP/M can handle up to four terminals; CP/M handles only one.

HOW THE CPU THINKS

WORD

The computer's memory is divided into chunks that are called *words*. On a PDP-10 computer, each word consists of 36 bits. Other computers have words that are longer or shorter:

COMPUTER	WORD LENGTH	
CDC (6000, 7000, Cyber 70, and Cyber 170)	60 bits	*maxicomputers*
PDP-10, PDP-20, and Honeywell 6000	36 bits	
IBM (360 and 370)	32 bits	
Nova, Eclipse, and PDP-11	16 bits	*minicomputers*
PDP-8	12 bits	
8080, 6800, 6502, Z-80, and F-8	8 bits	*microcomputers*

I'll begin by explaining the PDP-10; afterwards, I'll explain the others....

PDP-10 (and PDP-20) A FORTRAN integer is stored as one word. If you want to store the integer 74, which is 1001010 in binary, it will be stored like this:

000000000000000000000000001001010

Every positive integer must begin with a zero, so the largest binary integer that can be stored is—

011111111111111111111111111111111

—which in decimal notation is 34359738367. So in FORTRAN, if you try to write an integer larger than 34359738367, the computer will gripe and say OVERFLOW.

To store a negative integer, the computer does three steps:

1. Store its absolute value.
2. Find the rightmost 1.
3. Left of the rightmost 1, replace each 0 by 1, and each 1 by 0.

For example, to store -74, the computer does these steps:

1. Store 74, which is
   000000000000000000000000001001010
2. Find the rightmost 1.
   000000000000000000000000001001010
                                  ↑
3. Left of the rightmost 1, replace each 0 by 1, and each 1 by 0.
   111111111111111111111111110110110
   That word represents -74.

For a positive number, the word's leftmost bit is 0; but for a negative number, the leftmost bit is 1. Since the leftmost bit indicates the number's sign, it is called the *sign bit*.

If you ask the computer to add 77 to -74, here's what it does:

It stores 77 as 000000000000000000000000001001101
It stores -74 as 111111111111111111111111110110110
The binary sum is 1000000000000000000000000000000011

because 1+1=10

But this extra bit can't fit in a 36-bit word, so it is ignored. The word contains only 000000000000000000000000000000011, which is 3.

To store a positive real number, the computer does these steps:

1. Write it in binary, showing the decimal point.
2. Move the decimal point until it is immediately left of the leftmost 1. The new number is called the *mantissa*. Let E be how many places you moved the decimal point; if you moved the decimal point toward the left, E is positive; if you moved the decimal point toward the right, E is negative. (E is called the *exponent*.)
3. Write a 0, followed by eight bits for E+128, followed by twenty-seven bits for the mantissa.

For example, to store the real number 1.625, the computer does these steps....

1. Write 1.625 in binary:
   1.101
2. Move the decimal point, to get the mantissa:
   .1101
   Since the decimal point was moved 1 place, E is 1.
3. Write a 0, followed by eight bits for E+128 (which is 129), followed by twenty-seven bits for the mantissa:
   01000000111010000000000000000000000

   129 *.1101*

In step 2, if the computer moves the decimal point incorrectly, the number is called *unnormalized*. For example, if the computer moves the decimal point two places instead of one, the mantissa will be .01101 (instead of .1101), E will be 2 (instead of 1), and the number will be unnormalized: 01000001001101000000000000000000000.

130 *.01101*

Since the computer can do arithmetic even on numbers that are unnormalized, sometimes it doesn't bother to normalize the numbers.

To store the real number 5.0 (normalized), the computer must do these steps....

1. Write 5.0 in binary:
   101.0
2. Move the decimal point, to get the mantissa:
   .1010
   Since the decimal point was moved 3 places, E is 3.
3. Write a 0, followed by eight bits for E+128 (which is 131), followed by twenty-seven bits for the mantissa:
   01000001110100000000000000000000000

   131 *.1010*

By contrast, the integer 5 (which is binary 101) is stored simply as:
00000000000000000000000000000101
Since the method for storing 5 is simpler than for 5.0, the computer will perform 5+3 more quickly than 5.0+3.0. So to make your FORTRAN program run quickly, use integer arithmetic instead of real arithmetic, whenever possible.

In BASIC, unfortunately, all numbers are treated as if they were reals; 5 and 5.0 are both stored as:
01000001110100000000000000000000000
So BASIC "integer arithmetic" is actually real arithmetic, and hence slower than in FORTRAN.

The largest real number the computer can store is—
01111111111111111111111111111111111

E+128 *mantissa*

—which is about 1.7×10^{38}.

The computer negates a real number the same way it negates and integer. To store -5.0, the computer does these steps....

1. Store 5.0, which is
   01000001110100000000000000000000
2. Find the rightmost 1:
   01000001110100000000000000000000
3. Left of the rightmost 1, replace each 0 by 1, and each 1 by 0:
   10111110001100000000000000000000

To store a double-precision number, the computer figures out the mantissa more precisely than for a real. The double-precision number is stored as *two* words: the first word contains the sign bit, the exponent (+128), and the beginning of the mantissa; the second word is devoted mainly to the rest of the mantissa.

CDC (6000, 7000, Cyber 70, and Cyber 170) Since a word holds 60 bits instead of 36, CDC computers can handle larger integers and larger reals, and store more places after the decimal point than PDP-10 computers. The disadvantage of CDC computers is that they require more memory, since every number consumes 60 bits.

To negate a number, CDC computers replace *every* 0 by 1, and *every* 1 by 0. Even the bits on the right end of the word get replaced. The CDC scheme has several disadvantages; for example, it makes zero be
000000000000000000000000000000000000000000000000000000000000
but makes "negative zero" be different:
111111111111111111111111111111111111111111111111111111111111
To make negative zero differ from zero contradicts the laws of math. To do math correctly, CDC computers have to "fudge".

To store a positive real number, CDC computers write a 0, followed by eleven bits for E+976, followed by forty-eight bits for the mantissa.

IBM (360 and 370) A word holds 32 bits. An IBM integer resembles a PDP-10 integer but is 4 bits shorter.

To store a real, the computer uses hexadecimal. For example, to store 45.625, the computer does these steps....

1. Write 45.625 in binary:
   101101.101
   Then write it in hexadecimal:
   101101.101
   2 D . A
2. Put the decimal point left of the first non-zero digit, to get the mantissa:
   .2DA
   Since the decimal point was moved 2 places, E is 2.
3. Write a 0, followed by seven bits for E+64 (which is 66), followed by twenty-four bits for the manissa:
   01000010001011011010000000000000
   66 2 D A

PDP-11 A word holds 16 bits. To store a FORTRAN integer, the computer uses either one or two words, depending on the version of FORTRAN. To store a positive real, the computer writes a 0, followed by eight bits for E+128, followed by twenty-three bits for the "shortened mantissa". To form the shortened mantissa, omit the mantissa's leftmost bit (which is 1). For example, if the mantissa is .1101, the shortened mantissa is .101.

Other minicomputers and microcomputers Minicomputers and microcomputers have short words, because these computers were originally intended to handle only short integers. To store a longer integer or a real requires more than one word.

CHARACTER CODES

To store a character, the computer uses a 6-bit code, or 7-bit ASCII, or 8-bit ASCII, or 8-bit EBCDIC. Here are some calculations:

COMPUTER	WORD LENGTH	CHARACTER LENGTH	CONCLUSION (word length divided by char.)
CDC	60 bits	6 bits	A word can hold 10 characters.
PDP-10	36 bits	7 bits	A word can hold 5 characters. (The word's rightmost bit isn't used: it remains 0.)
IBM 360 & 370	32 bits	8 bits	A word can hold 4 characters.
PDP-11	16 bits	8 bits	A word can hold 2 characters.
PDP-8	12 bits	6 bits	A word can hold 2 characters.
microcomputers	8 bits	8 bits	A word can hold 1 character.

BCD

To handle numbers in COBOL programs, IBM 360 and 370 computers use this 4-bit code, called *Binary-Coded Decimal (BCD)*:

CHARACTER	BCD CODE	BCD CODE IN HEXADECIMAL
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
+	1100	C
-	1101	D

For example, the BCD code for the COBOL integer -928 is 1001 0010 1000 1101. The BCD code for the COBOL integer +21503 is 0010 0001 0101 0000 0011 1100. Although BCD was designed for COBOL, some microcomputers use it for BASIC.

Warning: if you read an old textbook that mentions "BCD", it might mean a 6-bit code for magnetic tapes, instead of the 4-bit code I've described.

KINDS OF LANGUAGES

The CPU doesn't understand BASIC, FORTRAN, PASCAL, or COBOL. The only language it understands is *machine language*, which is very primitive. A machine language program is just a list of zeros and ones, like this:

0001000010001000101010001011111011010001010111010111010110010110011000101110101110 1

In machine language, all numbers must be written in binary (not decimal); exponents, square roots, logarithms, and trigonometry are not allowed; keywords (such as IF and GO TO) must be written in a special binary code instead of in normal letters; and a variable's name must consist of just zeros and ones. On small computers, the machine language is *very* primitive: multiplication and division are not allowed; real numbers are not allowed (you must use only integers); what you print or input must be a string (not a number), and the string must be very short: it must have only one character! And instead of writing the character itself, you must write its ASCII or EBCDIC code. Writing a program in machine language is a pain in the ass.

On the main frame of a maxicomputer (or minicomputer or not-so-small microcomputer), you'll see a row of switches. By putting your fingers on the switches and flipping them up and down, you can feed bits into the computer. On most computers, a switch that's up represents a 1; a switch that's down represents a 0. The usual way to feed a machine-language program into the computer is to flip the switches. Since flipping a switch is called *toggling*, feeding a machine-language program into the computer by flipping switches is called *toggling the program in*.

To make the computer easier to program, the first step is to teach it *assembly language*, which resembles machine language but is slightly easier for a human to cope with, because:

- You can type an assembly-language program on a keyboard, instead of by toggling switc
- You can type the program in octal or hexadecimal instead of in binary.
- You can use variables, and give them names such as X and Y and PRICE.
- You can type commands such as ADD and AND, instead of having to type their binary cod

To teach the computer assembly language, store an explanation of assembly language on a disk (or in another part of the memory). The "explanation" is a program that makes the computer act as if it understands assembly language. The explanation-progra is called the *assembler* and is written in the only language the computer understands already: machine language.

The next step is to teach it languages such as BASIC, FORTRAN, and COBOL, by storing more explanation-programs (called *compilers* or *interpreters*), written in assembly language. BASIC, FORTRAN, and COBOL are called *high-level languages*. Machine languag and assembly language are *low-level languages*.

ELECTRONIC BOXES

From the CPU's point of view, the entire main frame is thousands of "electronic boxes" Each box holds several bits. Some of the boxes are long, and hold many bits. Other boxes are short, and hold few. Most of the boxes are in the main memory, and are called *memory locations*. A few of the boxes are in the CPU itself, and are called *registers*.

The CPU can move data from one box to another. Specifically, the CPU can do the following:

Move data from a memory location to a register. (That's called *loading* the data.)
Move data from a register to a memory location. (That's called *storing* the data.)
Move data from a register to another register.

The CPU can*not* move data from one memory location to another memory location directly. Instead, if you want to move data from one memory location to another, you must tell the CPU to move the data from the first memory location to a register, and then move the data from that register to the desired memory location.

The registers work more quickly than the memory locations. For example, moving data from a register to another register is faster than moving it to a memory location

The place the data is moved from is called the *source*. The place the data is moved to is called the *destination*. The source is a register or memory location, and so is the destination.

When data is "moved", it is merely copied. For example, when the CPU has "moved" data from a register to a memory location, the data is in both places: it is still in the register, and a copy of it is now in the memory location.

The memory locations are numbered in order: 0, 1, 2, 3, etc. So we speak of the "0th" memory location, the "1st" memory location, the "2nd" memory location, etc. A memory location's number is called the *address*. For example, the address of the 2nd memory location is: 2. The data the memory location contains is the *content*.

Pretend the main memory is a town, and the "electronic boxes" (memory locations) are the houses. Each house has an address. It also has content: the people inside it.

PROGRAM COUNTER

The CPU contains a special register, called the *program counter*. The program counter tells the CPU which line of your program to do next. If the program counter contains the number 6 (written in binary), the CPU will do the line of your program that's stored in the 6th memory location.

More precisely, here's what happens if the program counter contains the number 6:

A. The CPU moves the content of the 6th memory location to the CPU's *instruction register*. (That's called *fetching* the instruction.)

B. The CPU checks whether the instruction register contains a complete instruction written in machine language. If not—if the instruction register contains only *part* of a machine-language instruction—the CPU fetches the content of the 7th memory location also. (The instruction register is large enough to hold the content of memory locations 6 and 7 simultaneously.) If the instruction register still doesn't contain a complete instruction, the CPU fetches the content of the 8th memory location also. If the instruction register still doesn't contain a complete instruction, the CPU fetches the content of the 9th memory location also.

C. The CPU changes the number in the program counter. For example, if the CPU has fetched from the 6th and 7th memory locations, it makes the number in the program counter be 8; if the CPU has fetched from the 6th, 7th, and 8th memory locations, it makes the number in the program counter be 9. (That's called *updating* the program counter.)

D. The CPU figures out what the instruction means. (That's called *decoding* the instruction.)

E. The CPU obeys the instruction. (That's called *executing* the instruction.) If it's a "GO TO" type of instruction, the CPU makes the program counter contain the address of the memory location you want to go to.

After the CPU completes steps A, B, C, D, and E, it looks at the program counter and moves on to the next instruction. For example, if the program counter contains the number 9 now, the CPU does steps A, B, C, D, and E again, but by fetching, decoding, and executing the 9th memory location istead of the 6th.

The CPU repeats steps A, B, C, D, and E again and again; each time, the number in the program counter changes. Those five steps form a loop, called the *instruction cycle*.

ARITHMETIC/LOGIC UNIT

The CPU contains two parts: the *control unit* (which is the boss) and the *arithmetic/logic unit (ALU)*. When the control unit comes to step D of the instruction cycle, and decides some arithmetic or logic needs to be done, it sends the problem to the ALU, which sends back the answer.

Here's what the ALU can do:

NAME OF OPERATION	EXAMPLE	EXPLANATION
"plus", "added to", "+"	10001010 +10001001 answer: 100010011	add, but remember that 1+1 is 10 in binary
"minus", "subtract", "-"	10001010 -10001001 answer: 00000001	subtract, but remember that 10-1 is 1 in binary
"negative", "the two's complement of", "-"	-10001010 answer: 01110110	*left of the rightmost 1*, replace each 0 by 1, and each 1 by 0
"not", "the complement of", "the one's comp. of", "~"	NOT 10001010 answer: 01110101	replace each 0 by 1, and each 1 by 0
"and", "&", "∧"	10001010 AND 10001001 answer: 10001000	put a 1 wherever both original numbers had 1
"or", "inclusive or", "∨"	10001010 OR 10001001 answer: 10001011	put a 1 wherever some original number had 1
"eXclusive OR", "XOR", "⊻", "⊕"	10001010 XOR 10001001 answer: 00000011	put a 1 wherever the original numbers differ

Also, the ALU can shift a register's bits. For example, suppose a register contains 10111001. The ALU can shift the bits toward the right:

before 10111001
after 01011100

It can shift the bits toward the left:

before 10111001
after 01110010

It can rotate the bits toward the right:

before 10111001
after 11011100

It can rotate the bits toward the left:

before 10111001
after 01110011

It can shift the bits toward the right *arithmetically*:

before 10111001
after 11011100

It can shift the bits toward the left arithmetically:

before 10111001
after 11110010

Doubling a number is the same as shifting it left arithmetically. For example, doubling six (to get twelve) is the same as shifting six left arithmetically:

six 00000110
twelve 00001100

Halving a number is the same as shifting it right arithmetically. For example, halving six (to get three) is the same as shifting six right arithmetically:

six 00000110
three 00000011

Halving negative six (to get negative three) is the same as shifting negative six right arithmetically:

negative six 11111010
negative three 11111101

Using the ALU, the control unit can do operations such as

A. Find the number in the 6th memory location, and move its negative to a register.
B. Change the number in a register, by adding to it the number in the 6th memory location.
C. Change the number in a register, by subtracting from it the number in the 6th memory location.

On most computers, each operation has one source and one destination. In operations A, B, and C, the source is the 6th memory location; the destination is the register. The control unit cannot do a command such as "add together the number in the 6th memory location and the number in the 7th memory location, and put the sum in a register", because that operation would require two sources. Instead, you must give two shorter commands:

1. Move the number in the 6th memory location to the register.
2. Then add to that register the number in the 7th memory location.

FLAGS

The CPU contains a *flag register*, which comments on what the CPU is doing. In a typical CPU, the flag register has six bits, named as follows:
the Negative bit
the Zero bit
the Carry bit
the Overflow bit
the Priority bit
the Privilege bit

When the CPU performs an operation (such as addition, subtraction, shifting, rotating, or moving), the operation has a source and a destination; the number that goes into the destination is the operation's result. If the result is a negative number, the CPU turns on the Negative bit; in other words, it makes the Negative bit be 1. (Conversely, if the result is a number that's not negative, the CPU makes the Negative bit be 0.) If the result is zero, the CPU turns on the Zero bit; in other words, it makes the Zero bit be 1.

When the ALU computes the result, it also computes an extra bit, which becomes the Carry bit. For example, here's how the ALU adds 7 and -4:

```
7 is                    00000111
-4 is                   11111100
binary addition gives  100000011
                       ↑ result
                     Carry
```

So the result is 3, and the Carry bit becomes 1.

If the ALU can't compute a result correctly, it turns on the Overflow bit. For example, in elementary school you learned that 98+33 is 131; so in binary, the computation should look like this:

```
                128 64 32 16  8  4  2  1
98 is             1  1  0  0  0  1  0
33 is                1  0  0  0  0  1
the sum is      1 [0  0  0  0  0  1  1], which is 131
```

But here's what an 8-bit ALU will do:

```
              sign 64 32 16  8  4  2  1
98 is            0  1  1  0  0  0  1  0
33 is            0  0  1  0  0  0  0  1
the sum is  0    1  0  0  0  0  0  1  1
            ↑    [result              ]
          Carry
```

Unfortunately, the result's leftmost 1 is in the position marked *sign*, instead of the position marked 128; so the result looks like a negative number. To warn you that the result is incorrect, the ALU turns on the Overflow bit. If you're programming in a language such as BASIC, the interpreter or compiler keeps checking whether the Overflow bit is on; when it finds that the bit is on, it prints the word OVERFLOW.

While your program is running, it might be interrupted. Peripherals might interrupt, in order to input or output the data; the *real-time clock* might interrupt, to prevent you from hogging too much time, and to give another program a chance to run; and the computer's sensors might interrupt, when they sense that the computer is malfunctioning. When something wants to interrupt your program, the CPU checks whether your program has priority, by checking the Priority bit. If the Priority bit is on, your program has priority and cannot be interrupted.

On a computer that's handling several programs at the same time, some operations are dangerous: if your program makes the computer do those operations, the other programs might be destroyed. Dangerous operations are called *privileged instructions*; to use them, you must be a *privileged user*.

When you walk up to a terminal attached to a large computer, and type HELLO or LOGIN, and type your user number, the operating system examines your user number to find out whether you are a privileged user. If you are, the operating system turns on the Privilege bit. When the CPU starts running your programs, it refuses to do privileged instructions unless the Privilege bit is on.

Microcomputers omit the Privilege bit, and can't prevent you from giving dangerous commands. But since the typical microcomputer has only one terminal, the only person your dangerous command can hurt is yourself.

Some computers have *several* levels of priority and privilege. If your priority level is "moderately high", your program is immune from most interruptions, but not from all of them. If your privilege level is "moderately high", you can order the CPU to do most of the privileged instructions, but not all of them. To allow those fine distinctions, large computers devote *several* bits to explaining the priority level, and *several* bits to explaining the privilege level.

The bits in the flag register are called the *flags*. To emphasize that the flags comment on your program's status, people sometimes call them *status flags*.

In the CPU, the program counter is next to the flag register. Instead of viewing them as separate registers, some programmers consider them to be parts of a single big register, called the *program status word*.

You can give a command such as, "Test the 3rd memory location". The CPU will examine the number in the 3rd memory location; if that number is negative, the CPU will turn on the Negative bit; if that number is zero, the CPU will turn on the Zero bit.

You can give a command such as, "Test the difference between the number in the 3rd register and the number in the 4th". The CPU will adjust the flags according to whether the difference is negative or zero or carries or overflows.

You can use the flags, by giving a command such as, "If the Negative bit is on, go do the instruction in memory location 6".

SPEED

Computers are fast. To describe computer speeds, programmers use these words:

WORD	ABBREVIATION	MEANING
millisecond	msec *or* ms	thousandth of a second; 10^{-3} seconds
microsecond	μsec *or* μs	millionth of a second; 10^{-6} seconds
nanosecond	nsec *or* ns	billionth of a second; 10^{-9} seconds
picosecond	psec *or* ps	trillionth of a second; 10^{-12} seconds

1000 picoseconds is a nanosecond; 1000 nanoseconds is a microsecond; 1000 microseconc is a millisecond; 1000 milliseconds is a second.

On page 69, I said the *instruction cycle* has five steps:

A. Fetch the instruction.
B. Fetch additional parts for the instruction.
C. Update the program counter.
D. Decode the instruction.
E. Execute the instruction.

The total time to complete the instruction cycle is about a microsecond. The exact time depends on the quality of the CPU, the quality of the main memory, and the difficulty of the instruction, but usually lies between .1 microseconds and 10 microseconds.

Here are several ways to make the computer act more quickly:

METHOD	MEANING
multiprocessing	*The main frame holds more than one CPU.* All the CPUs work simultaneously. They share the same main memory. The operating system decides which CPU works on which program. The collection of CPUs is called a *multiprocessor.*
instruction lookahead	*While the CPU is finishing an instruction cycle (by doing steps D and E), it simultaneously begins working on the next instruction cycle* (steps A and B).
array processing	*The CPU holds at least 16 ALUs.* All the ALUs work simultaneously. For example, when the control unit wants to solve 16 multiplication problems, it sends each problem to a separate ALU; the ALUs compute the products simultaneously. The collection of ALUs is called an *array processor.*
parallel functional units	*The ALU is divided into several functional units:* an addition unit, a multiplication unit, a division unit, a shift unit, etc. All the units work simultaneously; while one unit is working on one problem, another unit is working on another.
pipeline architecture	*The ALU (or each ALU functional unit) consists of a "first stage" and a "second stage".* When the control unit sends a problem to the ALU, the problem enters the first stage, then leaves the first stage and enters the second stage. But while the problem is going through the second stage, a new problem starts going through the first stage. Such an ALU is called a *pipeline processor.*

PARITY

Most large computers put an extra bit at the end of each memory location. For example, a memory location in the PDP-10 holds 36 bits, but the PDP-10 puts an extra bit at the end, making 37 bits altogether. The extra bit is called the *parity bit.* If the number of ones in the memory location is even, the CPU turns the parity bit on; if the number of ones in the memory location is odd, the CPU turns the parity bit off. For example, if the memory location contains these 36 bits—

000000000100010000000110000000000000

there are 4 ones, so the number of ones is even, so the CPU turns the parity bit on:

000000000100010000000110000000000001

content parity

If the memory location contains these 36 bits instead—

000000000100010000000100000000000000

there are 3 ones, so the number of ones is odd, so the CPU turns the parity bit off:

000000000100010000000100000000000000

content parity

Whenever the CPU puts data into the main memory, it also puts in the parity bit. Whenever the CPU grabs data from the main memory, it checks whether the parity bit still matches the content. If the parity bit doesn't match, the CPU knows there was an error, and tries once again to grab the content and the parity bit. If the parity bit disagrees with the content again, the CPU decides that the memory is broken, refuses to run your program, prints a message saying PARITY ERROR, and then sweeps through the whole memory, checking the parity bit of every location; if the CPU finds another parity error (in your program or anyone else's), the CPU shuts off the whole computer.

That's how the PDP-10 protects itself against faulty memory. Most maxicomputers and large minicomputers use variations of that scheme. Small minicomputers and microcomputers don't have parity bits in the memory, and are less reliable.

UAL

Universal Assembly Language (UAL) is a notation I invented, that makes programming in assembly language easier.

UAL uses these symbols:

SYMBOL	MEANING
M0	the number in the 0th memory location
M1	the number in the 1st memory location
M2	the number in the 2nd memory location
etc.	
R0	the number in the 0th register
R1	the number in the 1st register
R2	the number in the 2nd register
etc.	
P	the number in the program counter
N	the Negative bit
Z	the Zero bit
C	the Carry bit
V	the oVerflow bit
PRIORITY	the PRIORITY bits
PRIVILEGE	the PRIVILEGE bits
F	the content of the entire Flag register
F[5]	the 5th bit in the Flag register
R2[5]	the 5th bit in R2
R2[LEFT]	the left half of R2; in other words, the left half of the data in the 2nd re
R2[RIGHT]	the right half of R2
M5 M6	a long number, whose left half is in the 5th memory location, and whose right half is in the 6th memory location
M5L2	same as M5 M6
M5L3	same as M5 M6 M7

Here are UAL statements:

STATEMENT	MEANING
R2=7	Let R2 be 7. In other words, let the number in the 2nd register be 7. In other words, move 7 into the 2nd register.
R2=M5	Let R2 be M5. In other words, let the number in the 2nd register be the number in the 5th memory location. In other words, move the 5th memory location's contents into the 2nd register.
R2= =M5	Exchange R2 with M5. In other words, put the 5th memory location's content into the 2nd register, and vice versa.
R2=R2+M5	Let R2 be R2 plus M5. In other words, change the integer in the 2nd register, by adding to it the integer in the 5th memory location.
R2=R2-M5	Subtract integers.
R2=R2*M5	Multiply integers.
R2 REM R3=R2/M5	Change the integer R2, by dividing into it the integer M5; and put the division's remainder into the 3rd register.
R2=R2+M5 REAL	Add real numbers.
R2=R2+M5 REAL UNNORM	Add real numbers, but don't bother to normalize.
R2=R2+M5 DECIMAL	Add BCD numbers.
R2=M5 TO REAL	Let R2 be M5 converted to a real (from an integer). In other words, let R2 be the real that has the same value as the integer M5.
R2=M5 TO INTEGER	Let R2 be M5 converted to an integer (from a real).
R2=M5 TO DECIMAL	Let R2 be M5 converted to a BCD number (from a binary integer).
R2=M5 TO BINARY	Let R2 be M5 converted to a binary integer (from a BCD number).
R2=M5 NORM	Let R2 be M5 normalized. In other words, let R2 be the normalize real that has the same value as the unnormalized real M5.
R2=-M5	Let R2 be the negative of M5.
R2=NOT M5	Let R2 be the one's complement of M5.
R2=R2 AND M5	Change R2, by performing the AND operation.
R2=R2 OR M5	Change R2, by performing the OR operation.
R2=R2 XOR M5	Change R2, by performing the XOR operation.

SHIFTL R2	Shift left.
SHIFTR R2	Shift right.
SHIFTRA R2	Shift right arithmetically.
SHIFTR3 R2	Shift right, 3 times.
SHIFTR(R7) R2	Shift right, R7 times.
ROTATEL	Rotate left.
ROTATER	Rotate right.
TEST R2	Test R2. In other words, examine the number in the 2nd register, and adjust the Negative and Zero bits.
TEST R2-R4	Test R2-R4. In other words, examine the difference between R2 and R4, and adjust the flag register.
CONTINUE	No operation; just continue on to the next instruction.
WAIT	Wait until an interrupt occurs.
IF R2<0, P=7	If R2<0, let P=7. In other words, if the number in the 2nd register is negative, move 7 into the program counter.
IF R2<0, M5=3, P=7	If R2<0, do both of the following: let M5 be 3, and P be 7.
DO M5	Do the statement in the 5th memory location. In other words, fetch, decode, and execute the statement in the 5th memory location, but without changing the number in the program counter.

M5 can be written as M(5) or M(2+3). It can be written as M(R7), if R7 is 5—other words, if register 7 contains 5.

A number is octal unless it is preceded by the symbol # (for hexadecimal).

ADDRESSING MODES

Suppose you want the 5th register to contain the number 6. You can accomplish that goal in one step, like this:

```
R5=6
```

Or you can accomplish it in two steps, like this:

```
M7=6
R5=M7
```

Or you can accomplish it in three steps, like this:

```
M7=6
M3=7
R5=M(M3)
```

Or you can accomplish it in an even weirder way:

```
M7=6
R2=4
R5=M(3+R2)
```

Each of those methods has a name. The first method (R5=6), which is the simplest, is called *immediate addressing*. The second method (R5=M7), which contains the letter M, is called *direct addressing*. The third method (R5=M(M3)), which contains the letter M twice, is called *indirect addressing*. The fourth method (R5=M(3+R2)), which contains the letter M and a plus sign, is called *indexed addressing*.

In each method, the 5th register is the destination. In the last three methods, the 7th memory location is the source. In the fourth method, which involves R2, the 2nd register is called the *index register*, and R2 itself is called the *index*.

Each of those methods is called an *addressing mode*. So you've seen four addressing modes: immediate, direct, indirect, and indexed.

To handle the program counter, the computer uses other addressing modes instead. For example, suppose P (the number in the program counter) is 2073, and you want to change it to 2077. You can accomplish that goal simply, like this:

```
P=2077
```

Or you can accomplish it in a weirder way, like this:

```
P=P+4
```

Or you can accomplish it in an even weirder way, like this:

```
R3=20
P=R3 77
```

The first method (P=2077), which is the simplest, is called *absolute addressing*. The second method (P=P+4), which involves addition, is called *relative addressing*; the "4" is the *offset*. The third method (P=R3 77) is called *base-page addressing*. R3 (which is 20) is called the *page number*; and so the 3rd register is called the *page register*.

THE CPU'S MAP

NINE MAPS

If one CPU is very similar to another, we say that both CPU's belong to the same *family*. In this chapter, we examine nine families:

Name of family	The most popular members of the family
PDP-8	PDP-8/A, PDP-8/E, Intersil 6100
PDP-10	PDP-10 (or Decsystem-10), PDP-20 (or Decsystem-20)
PDP-11	PDP-11, VAX-11*
CDC	CDC 6200, CDC 6400, CDC 6600, CDC 7600, Cyber 70, Cyber 170
IBM 360	IBM 630, 370, 4321, 4331, 4341, 3031, 3032, 3033, 3081, 3084
6800	Motorola 6800, Motorola 6809*, Motorola 68000*
6502	MOS Technology 6502, 6502C, 6510
8080	Intel 8080A, 8085, 80C85, 8086*, 80186*, 8088*
Z-80	Zilog Z-80, Z-80A, Z-80B, Z-8001*, Z-8002*

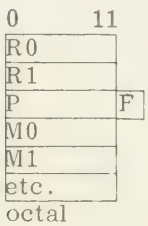
I've put an asterisk after each computer that's "souped up" by extra registers.

The first three families (PDP-8, PDP-10, and PDP-11) are all manufactured by Digital Equipment Corporation (DEC). The PDP-8 is a minicomputer. The PDP-10 is a maxicomputer. The PDP-11 is a "compromise" between the PDP-8 and PDP-10: it's larger than a PDP-8 but smaller than a PDP-10; it's a "large minicomputer".

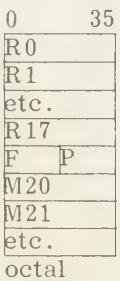
The next two families (CDC and IBM 360) are maxicomputers. The last four families (6800, 6502, 8080, and Z-80) are microcomputers.

Here's a *CPU map*, for each computer family:

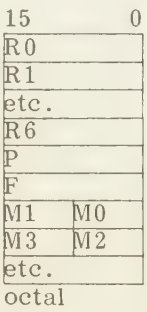
PDP-8



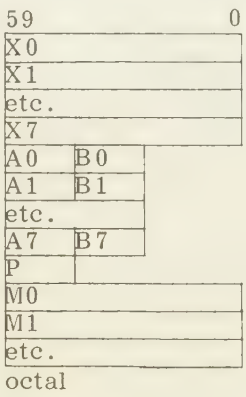
PDP-10



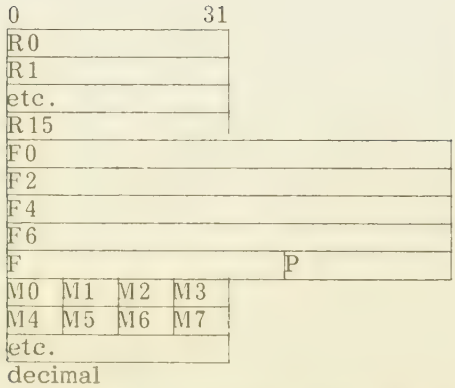
PDP-11



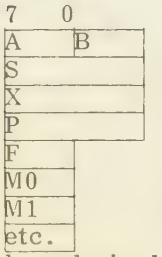
CDC



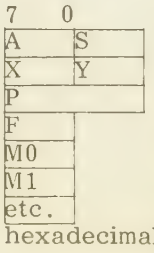
IBM 360



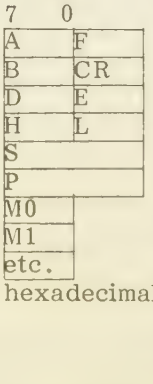
6800



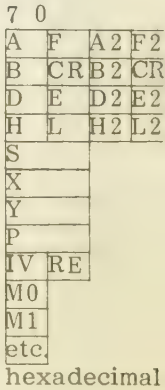
6502



8080



Z-80



Those maps are accurate for ordinary computers. Souped-up computers (which have asterisks) have fancier maps, which are not shown here.

PDP-8

To understand how to read a CPU's map, begin by looking at the map for the PDP-8, which is the simplest.

At the top of the map, you see the numbers 0 and 11. That means: in each word, the leftmost bit is called "bit 0", and the rightmost bit is called "bit eleven". So each word contains twelve bits, numbered from zero through eleven.

In the map, all the boxes have the same width, except for F's box, which is shorter. So R0, R1, P, M0, and M1 are each wide enough to hold a full word— twelve bits. F holds *less* than twelve bits. (In fact, F holds only 1 bit.)

R0 stands for register 0, and R1 stands for register 1, and P stands for the program counter, and M0 stands for memory location 0, and M1 stands for memory location 1. The map says that register 0, register 1, the program counter, memory location 0, and memory location 1 each hold a word. At the bottom of this map, the "etc." shows that the remaining memory locations (M2, M3, M4, etc.) each hold a word also. The flag register (represented by F) holds only 1 bit. (It's the Carry bit. Since the PDP-8 is cheap and crummy, it lacks other kinds of flags, such as Overflow, Priority, Privilege, Negative, and Zero.)

The general-purpose registers have names. Most books about the PDP-8 call R0 the "accumulator" and call R1 the "multiplier-quotient register".

At the bottom of the map, you see the word "octal". That means the memory locations are numbered octally. So the eighth memory location is called "M10" instead of "M8"; the ninth memory location is called "M11"; and the tenth memory location is called "M12".

PDP-10

The PDP-10 map shows that each word has thirty-six bits (numbered from 0 to 35). Each general-purpose register and each memory location holds thirty-six bits. The F and P are each a half-word; so F and P are each eighteen bits. (The word whose left half is F and whose right half is P is called the *program status word*.)

The flag register (F) includes an Overflow bit, Carry bit, Privilege bits, and other bits that describe overflows and carries in more detail. Four of the flag register's bits are *unused*: they remain 0. The PDP-10 does *not* have a Negative bit or a Zero bit. An auxiliary flag register (not shown) holds the Priority bits.

The bottom of the map says the numbering is octal. So there's no such thing as "M8" or "M9" or "R8" or "R9". The registers are numbered as follows:
R0 R1 R2 R3 R4 R5 R6 R7 R10 R11 R12 R13 R14 R15 R16 R17

Notice that the first memory location is called M20. If you try to make the CPU use a memory location numbered below M20 (such as M0 or M1 or M17), the CPU will use a general-purpose register instead. For example, if you tell the CPU to use M5, the CPU will use R5 instead.

PDP-11

In the PDP-11, each word contains sixteen bits, numbered from 0 to 15. Notice that "bit 0" is the bit on the far *right*. (On the PDP-8 and PDP-10, "bit 0" is the bit on the far *left*.)

The PDP-11 contains several general-purpose registers, numbered from R0 to R6. If you tell the CPU to use "R7", the CPU will use the program counter (P) instead.

Each memory location is just one byte, which is just eight bits, which is just half a word. So you need *two* memory locations to make a complete word. The first word of memory consists of M1 and M0; the next word of memory consists of M3 and M2; etc. W0 stands for the word M1 M0; W2 stands for the word M3 M2; W4 stands for the word M5 M4; etc.

CDC

In the CDC, each word is huge: sixty bits! Each memory location holds a word. The CPU contains general-purpose registers (X0 through X7); each holds a word. The CPU also contains a program counter (P), *address registers* (A0 through A7), and *index registers* (B0 through B7); each holds eighteen bits. B0 is always 0. The CPU lacks a flag register.

IBM 360

A word is thirty-two bits. Registers R0 through R15 each contain a word. Registers F0, F2, F4, and F6 are called the *floating point registers*; each contains two words. The flag register (F) contains 40 bits; the program counter (P) contains 24 bits.

Each memory location is just a byte, which is just eight bits, so you need four memory locations to hold a word. W0 stands for the word M0 M1 M2 M3; W4 stands for the word M4 M5 M6 M7; W8 stands for M8 M9 M10 M11; etc.

H0 stands for the halfword M0 M1; H2 stands for M2 M3; etc.

MICROCOMPUTERS

In each microcomputer family (6800, 6502, 8080, and Z-80), a word is just eight bits, which is just a byte. Each memory location is a word. The main general-purpose register is called the *accumulator (A)*, and is a word. The flag register (F) is a word. The program counter (P) is *two* words. The CPU also contains a register called the *stack pointer (S)*. Some of the microcomputers contain *index registers (X and Y)*.

Here are the flag register's bits:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
6800	1	1	HALFCARRY	PRIORITY	N	Z	V	C
6502	N	V	0	BREAK	DECIMAL	PRIORITY	Z	C
8080	N	Z	0	HALFCARRY	0	PARITY	1	C
Z-80	N	Z	0	HALFCARRY	0	V	SUBTRACT	C

On pages 71, 72, and 74, I explained the N, Z, C, V, and PRIORITY bits. In the 8080 and Z-80, the PRIORITY bit lies outside the flag register.

The BREAK bit helps handle interrupts. The PARITY bit helps do logic.

In the 6502, turning on the DECIMAL bit makes the CPU do arithmetic in the decimal system (BCD) instead of in binary. The 6800, 8080, and Z-80 have no DECIMAL bit, and can't do decimal arithmetic directly: if you want to do decimal arithmetic on those computers, you must tell the CPU to do the arithmetic in binary instead, and then give a command that says ADJUST. The HALFCARRY bit helps the computer ADJUST. The SUBTRACT bit helps the computer ADJUST a subtraction.

INSTRUCTION SETS

PDP-10

The list of instructions that a CPU understands is called the CPU's *instruction set*. Each instruction is written in machine language or assembly language.

To examine instruction sets, machine languages, and assembly languages, let's begin with the pleasantest CPU, which is the PDP-10. Here we go....

In PDP-10 machine language, each instruction is thirty-six bits. Here's an example:

010111100101100000010000000111001100

When you feed the CPU a machine-language instruction, the CPU begins by examining the three leftmost bits. If the bits are 111, the CPU will do something with the peripherals (such as output, input, or check on their status); the remaining bits explain which peripheral to use and which operation to perform. If the bits are 000 the CPU will use special software. If the bits are anything else, the instruction is normal. In the example above, the bits are 010, so the instruction is normal.

For normal instructions, the leftmost nine bits are called the *operation code* or *op code*:

010111100101100000010000000111001100

op code

The op code tells the CPU what kind of operation to do: addition, subtraction, multiplication, division, shifting, rotating, testing, or something weirder. In the example, the op code is 010111100, which means subtraction. It says to change a register's content, by subtracting the content of a memory location. But the op code is vague; it doesn't say *which* register and memory location to use. In other words, the op code says to do Ri=Ri-Mj, but doesn't say what i and j are.

To find i, the CPU looks at the next four bits:

010111100101100000010000000111001100

op code i

Since i is binary 1011, which in octal is 13, the CPU knows to do R13=R13-Mj. The

1011

only problem left is to find j.

The CPU tentatively assumes j is the right half of the word:

010111100101100000010000000111001100

op code i j?

So it assumes j is 010000000111001100, which in octal is 200714. But before making a final decision, the CPU checks whether the mode bits between i and j are zero:

010111100101100000010000000111001100

op code i mode j?

Since the mode bits are indeed zero, j is indeed 200714, and the CPU does R13=R13-M200714.

So to make the computer do R13=R13-M200714, you can feed it this machine-language instruction:

010111100101100000010000000111001100

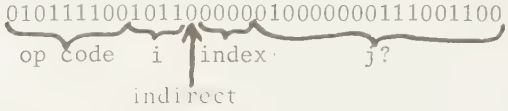
op code i mode j?

An easier way is to feed it this assembly-language instruction:

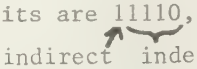
SUB 13,200714

The assembler will automatically translate the SUB into 010111100, the 13 into 1011, the comma into 00000, and the 200714 into 010000000111001100. That is, the assembler will translate octal into binary, and translate *mnemonics* (such as SUB) into op codes (such as 010111100).

The first mode bit is called the *indirect bit*; the remaining mode bits are called the *index bits*:



If the indirect bit is 1 instead of 0, the CPU does indirect addressing: instead of doing R13=R13-M200714, it does R13=R13-M(M200714). If the index bits are not 0000, the CPU does indexed addressing: for example, if the index bits are binary 1110, which is octal 16, the CPU does R13=R13-M(200714+R16). If the mode bits are 11110,



the CPU does indirect and also indexed addressing, like this:
R13=R13-M(M(200714+R16)). Here's how to write it in assembly language:

```
SUB 13,@200714(16)
```

indirect — index register

Besides subtraction, the CPU can do many other operations. Here are the op codes, their mnemonics, and their speeds:

	OP CODE	MNEMONIC	WHAT THE MNEMONIC STANDS FOR	UAL	MICROSECONDS (WITHOUT FETC
double precision	110	DFAD	Double Floating ADd	RiL2=RiL2+MjL2 REAL	2.59-7.
	111	DFSB	Double Floating SuBtract	RiL2=RiL2-MjL2 REAL	2.59-7.
	112	DFMP	Double Floating MultiPly	RiL2=RiL2*MjL2 REAL	6.89-10
	113	DFDV	Double Floating DiVide	RiL2=RiL2/MjL2 REAL	14.88-15
	120	DMOVE	Double MOVE	RiL2=MjL2	1.73
	121	DMOVN	Double MOVe Negative	RiL2=-MjL2	2.07-2.
	122	FIX	FIX	Ri=Mj TO INTEGER	1.72-3.
	124	DMOVEM	Double MOVE to Memory	MjL2=RiL2	1.85
	125	DMOVNM	Double MOVe Negative to Memory	MjL2=-RiL2	2.31
	126	FIXR	FIX and Round	Ri=Mj TO INTEGER ROUNDED	1.72-3.
	127	FLTR	FLoaT and Round	Ri=Mj TO REAL	2.10-6.
	130	UFA	Unnormalized Floating Add	R(i+1)=Ri+Mj REAL UNNORM	1.91-3.
	131	DFN	Double Floating Negate	Ri Mj=-(Ri Mj)	1.50
	132	FSC	Floating SCale	Ri=Ri*2↑Mj REAL	1.02-1.
	(133-137 move bytes instead of words)				
reals	140	FAD	Floating ADd	Ri=Ri+Mj REAL	2.45-6.
	141	FADL	Floating ADd Long	RiL2=Ri+Mj REAL	2.79-6
	142	FADM	Floating ADd to Memory	Mj=Ri+Mj REAL	2.80-6
	143	FADB	Floating ADd to Both	Ri=Ri+Mj REAL Mj=Ri	2.80-6
	144	FADR	Floating ADd and Round	Ri=Ri+Mj REAL ROUNDED	2.45-6
	145	FADRI	Floating ADd and Round Immediate	Ri=Ri+j REAL ROUNDED	1.75-5
	146	FADRM	Floating ADd and Round to Memory	Mj=Ri+Mj REAL ROUNDED	2.80-6
	147	FADRB	Floating ADd and Round to Both	Ri=Ri+Mj REAL ROUNDED Mj=Ri	2.80-6
	150	FSB	Floating SuBtract	Ri=Ri-Mj REAL	2.62-6
	(150-157, 160-167, and 170-177 resemble 140-147)				
	160	FMP	Floating MultiPly	Ri=Ri*Mj REAL	3.65-4
	170	FDV	Floating DiVide	Ri=Ri/Mj REAL	7.12-7
move a word	200	MOVE	MOVE	Ri=Mj	1.26
	201	MOVEI	MOVE Immediate	Ri=j	.45
	202	MOVEM	MOVE to Memory	Mj=Ri	1.06
	203	MOVES	MOVE to Self	Mj=Mj	1.61
	204	MOVS	MOVe Swapped	Ri=Mj[RIGHT] Mj[LEFT]	1.26
	(204-207, 210-213, 214-217, 500-503, and 504-507 resemble 200-203)				
	210	MOVN	MOVe Negative	Ri=-Mj	1.32
	214	MOVM	MOVe Magnitude	IF Mj>=0, Ri=Mj	1.32
				IF Mj<0, Ri=-Mj	1.32

multiply & divide integers	220 IMUL	Integer MULtiply	Ri=Ri*Mj	3.47-6.38
	(220-223, 224-227, 230-233, 234-237, 270-273, 274-277, 400-403, 404-407, 410-413, 414-417, 420-423, 424-427, 430-433, 434-437, 440-443, 444-447, 450-453, 454-457, 460-463, 464-467, 470-473, and 474-477 resemble 144-147)			
	224 MUL	MULtiply	Ri R(i+1)=Ri*Mj	3.63-7.14
	230 IDIV	Integer DIVide	Ri REM R(i+1)=Ri/Mj	8.16-8.45
	234 DIV	DIVide	Ri REM R(i+1)=(Ri R(i+1))/Mj	8.10-8.51
shift, rotate, & count	240 ASH	Arithmetic SHift	IF j>0, SHIFTLAj Ri IF j<0, SHIFTRA(-j) Ri	1.14-5.10
	241 ROT	ROTate	IF j>0, ROTATELj Ri IF j<0, ROTATER(-j) Ri	1.14-5.10
	242 LSH	Logical SHift	IF j>0, SHIFTLj Ri IF j<0, SHIFTR(-j) Ri	1.14-5.10
	243 JFFO	Jump if Find First One IF Ri<>0, R(i+1)=(HOW MANY LEADING ZEROS IN Ri), P=j	IF Ri=0, R(i+1)=0	.79-2.66
	244 ASHC	Arithmetic SHift Combined IF j>0, SHIFTLj Ri R(i+1)[1 TO 35] IF j<0, SHIFTRA(-j) Ri R(i+1)[1 TO 35]	IF j<>0, R(i+1)[0]=Ri[0]	1.14-9.06
	(244-246 resemble 240-242)			
fancy move	250 EXCH	EXCHange	Ri=:=Mj	1.61
	251 BLT	BLock Transfer M(Ri[RIGHT])L(LENGTH)=M(Ri[LEFT])L(LENGTH)	[illegible]	1.55+1.5[illegible]
	252 AOBJP	Add One to Both halves and Jump if Positive	Ri=(Ri[LEFT]+1) (Ri[RIGHT]+1) IF Ri>=0, P=j	.57
	253 AOBJN	Add One to Both halves and Jump if Negative	Ri=(Ri[LEFT]+1) (Ri[RIGHT]+1) IF Ri<0, P=j	.57
	254 JRST	Jump and ReStore	F=(DEPENDS ON i) P=j	.34
	255 JFCL	Jump on Flag and Clear IF (F[0 TO 3] AND i)<>0, F=(F[0 TO 3] AND NOT i), P=j		.34
	256 XCT	eXeCuTe	DO Mj	.34+(Mj TIME)
	257 MAP	MAP	Ri=j MAPPED	1.12
push & pop	260 PUSHJ	PUSH and Jump	Ri=(Ri[LEFT]+1) (Ri[RIGHT]+1) M(Ri[RIGHT])=F P P=j	1.12
	261 PUSH	PUSH	Ri=(Ri[LEFT]+1) (Ri[RIGHT]+1) M(Ri[RIGHT])=Mj	1.94
	262 POP	POP	Mj=M(Ri[RIGHT]) Ri=(Ri[LEFT]-1) (Ri[RIGHT]-1)	2.16
	263 POPJ	POP and Jump	Ri=(Ri[LEFT]-1) (Ri[RIGHT]-1) P=M(Ri[RIGHT]+1)	1.43
jump to & from subroutine	264 JSR	Jump to SubRoutine	Mj=F P P=j+1	.95
	265 JSP	Jump and Save P	Ri=F P P=j	.45
	266 JSA	Jump and Save Accumulator	Mj=Ri Ri=j P P=j+1	1.06
	267 JRA	Jump and Restore Accumulator	Ri=M(Ri[LEFT]) P=j	1.59
add & subtract integers	270 ADD	ADD	Ri=Ri+Mj	1.32
	274 SUB	SUBtract	Ri=Ri-Mj	1.32

maybe skip or jump

300	CAI	Compare Acc. Immed. & skip never	CONTINUE	.62
301	CAIL	Compare Acc. Im. & skip if Less	IF Ri<j, P=P+1	.62
302	CAIE	Compare Acc. Im. & skip if Equal	IF Ri=j, P=P+1	.62
303	CAILE	Compare Acc. Im. & skip if L.E.	IF Ri<=j, P=P+1	.62
304	CAIA	Compare Acc. Im. & skip Always	P=P+1	.62
305	CAIGE	Compare Acc. Im. & skip if G.E.	IF Ri>=j, P=P+1	.62
306	CAIN	Compare Acc. Im. & skip if N.	IF Ri<>j, P=P+1	.62
307	CAIG	Compare Acc. Im. & skip if Grea.	IF Ri>j, P=P+1	.62
(310-317, 320-327, 330-337, 340-347, 350-357, 360-367, and 370-377 resemble 300-307)				
311	CAML	Co. Acc. with Mem. & skip if Le.	IF Ri<Mj, P=P+1	1.43
321	JUMPL	JUMP if Less	IF Ri<0, P=j	.56
331	SKIPL	SKIP if Less	IF i<>0, Ri=Mj IF Mj<0, P=P+1	1.37
341	AOJL	Add One and Jump if Less	Ri=Ri+1 IF Ri<0, P=j	.62
351	AOSL	Add One and Skip if Less	Mj=Mj+1 IF i<>0, Ri=Mj IF Mj<0, P=P+1	1.78
361	SOJL	Subtract One and Jump if Less	Ri=Ri-1 IF Ri<0, P=j	.62
371	SOSL	Subtract One and Skip if Less	Mj=Mj-1 IF i<>0, Ri=Mj IF Mj<0, P=P+1	1.78

AND, OR, XOR, & NOT

400	SETZ	SET to Zeros	Ri=0	.45
404	AND	AND	Ri=Ri AND Mj	1.26
410	ANDCA	AND Complement of Accumulator	Ri=(NOT Ri) AND Mj	1.26
414	SETM	SET to Memory	Ri=Mj	1.26
420	ANDCM	AND Complement of Memory	Ri=Ri AND (NOT Mj)	1.26
424	SETA	SET to Accumulator	Ri=Ri	.45
430	XOR	XOR	Ri=Ri XOR Mj	1.26
434	IOR	Inclusive OR	Ri=Ri OR Mj	1.37
440	ANDCB	AND with Complements of Both	Ri=(NOT Ri) AND (NOT Mj)	1.26
444	EQV	EQuiValence	Ri=NOT (Ri XOR Mj)	1.26
450	SETCA	SET to Complement of Accumulator	Ri=NOT Ri	.45
454	ORCA	OR Complement of Accumulator	Ri=(NOT Ri) OR Mj	1.37
460	SETCM	SET to Complement of Memory	Ri=NOT Mj	1.26
464	ORCM	OR Complement of Memory	Ri=Ri OR (NOT Mj)	1.37
470	ORCB	OR Complements of Both	Ri=(NOT Ri) OR (NOT Mj)	1.37
474	SETO	SET to Ones	Ri=777777777777	.45

move half a word

500	HLL	Half word Left to Left	Ri[LEFT]=Mj[LEFT]	1.26
504	HRL	Half word Right to Left	Ri[LEFT]=Mj[RIGHT]	1.26
510	HLLZ	Half word Left to Left Zeros	Ri=Mj[LEFT] 000000	1.26
(510-517, 520-527, and 530-537 resemble 500-507)				
520	HLLO	Half word Left to Left Ones	Ri=Mj[LEFT] 777777	1.26
530	HLLE	Half word Left to L. Extend sign IF Mj[LEFT]>=0, Ri=Mj[LEFT] 000000 IF Mj[LEFT]<0, Ri=Mj[LEFT] 777777		1.26
540	HRR	Half word Right to Right	Ri[RIGHT]=Mj[RIGHT]	1.26
(540-577 resemble 500-537)				

create a mask

```
600 TRN    Test Right No modification       MASK=Ri[RIGHT] AND Mj                .34
601 TLN    Test Left No modification        MASK=Ri[LEFT] AND Mj                 .45
602 TRNE   Test R. No mod. & skip if Eq. z. MASK=Ri[RIGHT] AND Mj                .51
                                            IF MASK=0, P=P+1
(602-603, 604-605, and 606-607 resemble 601-602)
604 TRNA   Test R. No mod. & skip Always    MASK=Ri[RIGHT] AND Mj                .45
                                            P=P+1
606 TRNN   Test R. No mod. & skip if Not z. MASK=Ri[RIGHT] AND Mj                .51
                                            IF MASK<>0, P=P+1
610 TDN    Test Direct No modification      MASK=Ri AND Mj                      1.15
611 TSN    Test Swapped No modification     MASK=Ri AND (Mj[RIGHT] Mj[LEFT])    1.26
(610-617 resemble 600-607)
620 TRZ    Test Right and set to Zeros      MASK=Ri[RIGHT] AND Mj                .45
                                            Ri[RIGHT]=Ri[RIGHT] AND NOT MASK
(620-637, 640-657, and 660-677 resemble 600-617)
640 TRC    Test Right and set to Complement MASK=Ri[RIGHT] AND Mj                .45
                                            Ri[RIGHT]=Ri[RIGHT] XOR MASK
660 TRO    Test Right and set to Ones       MASK=Ri[RIGHT] AND Mj                .56
                                            Ri[RIGHT]= Ri[RIGHT] OR MASK
```

In that table, look at line 274. It says:

To make the computer do op code 274, your assembly-language program should contain the word SUB, which stands for SUBtract. Op code 274 makes the computer do "Ri=Ri-Mj" in 1.32 microseconds.

Look at line 220. It says:

To make the computer do op code 220, your assembly-language program should contain the word IMUL, which stands for Integer MULtiply. Op code 220 makes the computer do "Ri=Ri*Mj" in from 3.47 to 6.38 microseconds. (The exact time depends on what Ri and Mj are.)

The times in the table assume the addressing mode is immediate or direct. For indirect addressing, add 1.02 microseconds; for indexed addressing, add .06 microseconds.

Look at page 69 again—steps A, B, C, D, and E—the five steps the CPU must do to execute a machine-language instruction. The times in the table include just steps B, C, D, and E. The times do not include step A, which takes .89 microseconds. So to find the total time for op code 274, add .89 microseconds to the time in the table (1.32 microseconds), which gives you a total of 2.21 microseconds. The table omits the .89 microseconds because the CPU has instruction lookahead—it usually does step A of a new instruction at the same time as steps D and E of the previous instruction.

The PDP-10 family consists of four models: the PDP-KA10, the PDP-KI10, the PDP-KL10, and the PDP-20. The PDP-KA10 is the slowest. Most programs run twice as quickly on the PDP-KI10, and ten times as quickly on the PDP-KL10. The PDP-20 is about as fast as the PDP-KI10 but has a different operating system. The times in the table are for the PDP-KI10. (The PDP-KA10 lacks some of the op codes, lacks instruction lookahead, and has a slower ALU. The PDP-20 and PDP-KI10 have extra op codes, and the PDP-KL10 grabs information from memory more quickly.)

Here's a complete program, written in assembly language:

```
START:  MOVEI 3,647
        SUBI 3,2
        EXIT
        END START
```

The first line makes the computer do R3=647. The next line makes the computer do R3=R3-2, so R3 becomes 645. The EXIT and END START mark the end of the program.

When the program has finished executing, the 3rd register contains 645. To prove so, type this on your terminal:

```
E 3
```

That makes the operating system Examine the 3rd register. The operating system will print on your terminal:

```
000003/ 000000 000645
```

That means: the 3rd register contains 645.

The PDP-8 is smaller, simpler, and cheaper than the PDP-10.

In PDP-8 machine language, each instruction is just twelve bits, like this:

001000101011

In octal, that becomes:

001000101011

1 0 5 3

The leftmost octal digit is the op code:

1 0 5 3

op code

Here's what the op code means:

OP CODE	MNEMONIC	WHAT THE MNEMONIC STANDS FOR	UAL	MICROSECONDS
0	AND	AND	R0=R0 AND Mj	3
1	TAD	Twos complement ADd	R0=R0+Mj	3
2	ISZ	Increment and Skip if Zero	Mj=Mj+1 IF Mj=0, P=P+1	3
3	DCA	Deposit and Clear Accumulator	Mj=R0 R0=0	3
4	JMS	JuMp to Subroutine	Mj=P P=j+1	3
5	JMP	JuMP	P=j	1.5

(6 manipulates the peripherals. 7 manipulates just the CPU registers.)

In that table, the time includes all five steps (A, B, C, D, and E), and assumes the CPU is a PDP-8/A or PDP-8/I. (The PDP-8/E and PDP-8/M are slightly faster; the PDP-8/L and PDP-8/S are slower.)

The CPU tentatively assumes j is the rightmost seven bits:

001000101011

op j?

code

But before making a final decision, the CPU checks whether the mode bits are zero:

001000101011

op j?

code

mode

In that example, the op code is octal 1, which according to the table means R0=R0+Mj. Since the mode bits are zero, j is indeed 0101011, which is octal 053, so the CPU will do R0=R0+M053.

The first mode bit is called the *indirect bit*; the other mode bit is called the *base-page bit*:

001000101011

op j?

code

indirect? base-page?

Here's what happens if you change the indirect and base-page bits:

INDIRECT BIT	BASE-PAGE BIT	WHAT THE CPU WILL DO
0	0	R0=R0+M053
1	0	R0=R0+M(M053)
0	1	R0=R0+M(P[0 TO 4] 053)
1	1	R0=R0+M(M(P[0 TO 4] 053))

If the indirect bit is 1, and the base-page bit is 0, and the tentative j is at least 010 but less than 020 (instead of being 053), the CPU will do Mj=Mj+1 before doing the instruction.

PDP-11

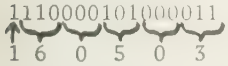
The PDP-11 is a compromise between the PDP-8 and the PDP-10. The PDP-8 is cheap but not powerful; the PDP-10 is powerful but not cheap; the PDP-11 is almost as cheap as the PDP-8, yet almost as powerful as the PDP-10, and is the most admired minicomputer in the world. You can get either the standard PDP-11 or the "souped-up" PDP-11. The souped-up version is called the VAX-11; it's more expensive. I'll discuss just the standard version here.

In PDP-11 machine language, each instruction is usually sixteen bits, like this:

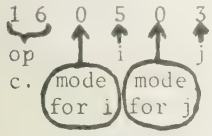
1110000101000011

In octal, that becomes:

1110000101000011



The leftmost pair of octal digits is the op code. Here's how the other octal digits are named:



If the mode for i and the mode for j are both 0, here's what the op code means:

OP CODE	ASSEMBLY LANGUAGE	WHAT THE MNEMONIC STANDS FOR	UAL
01	MOV Ri,Rj	MOVe	Rj=Ri
02	CMP Ri,Rj	CoMPare	TEST Ri-Rj
03	BIT Ri,Rj	BIt Test	TEST Ri AND Rj
04	BIC Ri,Rj	BIt Clear	Rj=Rj AND NOT Ri
05	BIS Ri,Rj	BIt Set	Rj=Rj OR Ri
06	ADD Ri,Rj	ADD	Rj=Rj+Ri
11	MOVB Ri,Rj	MOVe Byte	Rj[RIGHT]=Ri[RIGHT]
(11-15 resemble 01-05)			
16	SUB Ri,Rj	SUBtract	Rj=Rj-Ri

Here's how long those instructions take....

PDP-11/03: 3.5 μs always
PDP-11/04: 2.9 μs for op codes 01-03; 3.2 μs for codes 04-06 & 16; 2.9-3.9 for others
PDP-11/34: 1.8 μs for op codes 01 & 11; 2.0 μs for others (assuming the memory is core)
PDP-11/40: .9 μs for op code 01; 1.8 μs for code 11; 1.0 μs for others
PDP-11/45: 1.0 μs if j≠7; 1.2 μs if j=7 (assuming the memory is large core)
PDP-11/70: .3 μs if j≠7; .6 μs if j=7 (assuming the memory is core plus "cache")

If the modes are *not* zero, the instruction is more complicated and consumes more time. For example, here's what happens if the op code is 03, and the mode for i stays 0, but the mode for j changes:

MODE FOR j	MODE'S NAME	ASSEMBLY LANGUAGE	UAL
0	direct	BIT Ri,Rj	TEST Ri AND Rj
1	indirect	BIT Ri,@Rj	TEST Ri AND W(Rj)
2	autoincrement	BIT Ri,(Rj)+	TEST Ri AND W(Rj) Rj=Rj+2
3	autoincrement indirect	BIT Ri,@(Rj)+	TEST Ri AND W(W(Rj)) Rj=Rj+2
4	autodecrement	BIT Ri,-(Rj)	Rj=Rj-2 TEST Ri AND W(Rj)
5	autodecrement indirect	BIT Ri,@-(Rj)	Rj=Rj-2 TEST Ri AND W(W(Rj))
6	indexed	BIT Ri,k(Rj)	TEST Ri AND W(k+Rj) *(the instruction takes two words instead of one; the second word contains k)*
7	indexed indirect	BIT Ri,@k(Rj)	TEST Ri AND W(W(k+Rj)) *(the instruction takes two words instead of one; the second word contains k)*

The same modes are available for i. Here are examples:

MODE FOR i	MODE FOR j	ASSEMBLY LANGUAGE	UAL
1 (indirect)	0 (direct)	BIT @Ri,Rj	TEST W(Ri) AND Rj
7 (indexed)	6 (indexed indirect)	BIT n(Ri),@k(Rj)	TEST W(n+Ri) AND W(W(k+Rj)) *(the instruction takes three wo instead of one; the second word contains n; the third word cont k)*

For op codes 11-15 (which involve bytes instead of words), the modes have different meanings: for modes 1, 2, 4, and 6, replace the "W" by "M"; for modes 3, 5, and 7, replace the "W(W" by "M(W"; for modes 2 and 4, replace the "2" by "1".

Op codes 00, 07, and 10 do fancy logic. Op code 17, which is only on the PDP-11/34 and on models larger than the PDP-11/40, handles reals.

CDC

Who makes the fastest large family of computers? The answer is: Control Data Corporation. In 1965, it began selling the computer that made it famous: the CDC 6600, which can add together two numbers with extreme accuracy (60 bits) in .3 microseconds. Later, CDC began selling cheaper versions (the 6200 and 6400) and a more expensive version (the 7600):

MODEL	ADDING TWO 60-BIT INTEGERS	MULTIPLYING TWO 60-BIT REALS
6200	.9 μs	6.0 μs
6400	.6 μs	5.7 μs
6600	.3 μs	1.0 μs
7600	.055 μs	.138 μs

Although CDC computers were excellent at handling binary arithmetic, they were poor at handling strings. So CDC created a new series of computers (called the Cyber which includes a new string-handling unit (called the Compare/Move Unit, or CMU):

MODEL	DEFINITION
Cyber 70/72	6200 with CMU
Cyber 70/73	6400 with CMU
Cyber 70/74	6600 with CMU
Cyber 70/76	7600 with CMU

In 1974, CDC began selling an even faster series, called the CDC Cyber 170.

In CDC machine language, each instruction begins with a six-bit op code, followed by a three-bit i, a three-bit j, and a k. Usually k has three bits; but for op codes 00-07, 50-52, 60-62, and 70-72, it has eighteen bits; and for op code 46, it has either three or eighteen or forty-eight bits, depending on i.

CDC's assembly language is called COMPASS. Like PDP assembly languages, COMPASS uses octal. But instead of using mnemonics, COMPASS uses symbols such as "+" and "*":

	OP CODE	COMPASS SYMBOL	WHAT THE SYMBOL STANDS FOR	UAL	MICROSECONDS 70/72 .1-∞	70/73 .1-∞	70/74 .1-∞
	00	PS	Program Stop	WAIT			
	01 handles subroutines, "[illegible]"						
	02	JP Bi+k	JumP	P=Bi+k	1.6	1.3	1.4
	03 makes the computer examine i and Xj and maybe do P=k)						
branch	04	EQ Bi,Bj,k	EQual	IF Bi=Bj, P=k	1.6	.5-1.3	.8-1.2
branch	05	NE Bi,Bj,k	Not Equal	IF Bi<>Bj, P=k	1.6	.5-1.3	.8-1.2
branch	06	GE Bi,Bj,k	Greater or Equal	IF Bi>=Bj, P=k	1.6	.5-1.3	.8-1.2
branch	07	LT Bi,Bj,k	Less Than	IF Bi<=Bj, P=k	1.6	.5-1.3	.8-1.2
logic	10	BXi Xj	Boolean logic	Xi=Xj	.8	.5	.3
logic	11	BXi Xj*Xk	Boolean logic	Xi=Xj AND Xk	.8	.5	.3
logic	12	BXi Xj+Xk	Boolean logic	Xi=Xj OR Xk	.8	.5	.3
logic	13	BXi Xj-Xk	Boolean logic	Xi=Xj XOR Xk	.8	.5	.3
logic	14	BXi -Xk	Boolean logic	Xi=NOT Xk	.8	.5	.3
logic	15	BXi -Xk*Xj	Boolean logic	Xi=Xj AND NOT Xk	.8	.5	.3
logic	16	BXi -Xk+Xj	Boolean logic	Xi=Xj OR NOT Xk	.8	.5	.3
logic	17	BXi -Xk-Xj	Boolean logic	Xi=Xj XOR NOT Xk	.8	.5	.3
shift & rotate	20	LXi jk	Left	ROTATELjk Xi	.9	.6	.3
shift & rotate	21	AXi jk	Arithmetic shift	SHIFTRAjk Xi	.9	.6	.3
shift & rotate	22	LXi Bj,Xk	Left	Xi=Xk ROTATEL(Bj) Xi	.9	.6	.3
shift & rotate	23	AXi Bj,Xk	Arithmetic shift	Xi=Xk SHIFTRA(Bj) Xi	.9	.6	.3
normalize	24	NXi Bj,Xk	Normalize	Xi COUNT Bj=Xk NORM	1.0	.7	.4
normalize	25	ZXi Bj,Xk	Znormalize	Xi COUNT Bj=Xk NORM ROUNDED	1.0	.7	.4
	26	UXi Bj,Xk	Unpack	Bj PACKED WITH Xi=Xk	1.0	.7	.3
	27	PXi Bj,Xk	Pack	Xi=Bj PACKED WITH Xk	1.0	.7	.3
add & subtract	30	FXi Xj+Xk	Floating point	Xi=Xj+Xk REAL	1.4	1.1	.4
add & subtract	31	FXi Xj-Xk	Floating point	Xi=Xj-Xk REAL	1.4	1.1	.4
add & subtract	32	DXi Xj+Xk	Double precision	Xi=(Xj+Xk REAL DOUBLE)[RIGHT]	1.4	1.1	.4
add & subtract	33	DXi Xj-Xk	Double precision	Xi=(Xj-Xk REAL DOUBLE)[RIGHT]	1.4	1.1	.4
add & subtract	34	RXi Xj+Xk	Round	Xi=Xj+Xk REAL ROUNDED	1.4	1.1	.4
add & subtract	35	RXi Xj-Xk	Round	Xi=Xj-Xk REAL ROUNDED	1.4	1.1	.4
add & subtract	36	IXi Xj+Xk	Integer	Xi=Xj+Xk	.9	.6	.3
add & subtract	37	IXi Xj-Xk	Integer	Xi=Xj-Xk	.9	.6	.3
multiply	40	FXi Xj*Xk	Floating point	Xi=Xj*Xk REAL	6.0	5.7	1.0
multiply	41	RXi Xj*Xk	Round	Xi=Xj*Xk REAL ROUNDED	6.0	5.7	1.0
multiply	42	DXi Xj*Xk	Double precision	Xi=(Xj*Xk REAL DOUBLE) RIGHT	6.0	5.7	1.0
	43	MXi jk	Mask	Xi=(jk ONES) (60-jk ZEROS)	.9	.6	.3
divide	44	FXi Xj/Xk	Floating point	Xi=Xj/Xk REAL	6.0	5.7	2.9
divide	45	RXi Xj/Xk	Round	Xi=Xj/Xk REAL ROUNDED	6.0	5.7	2.9
	46 handles strings, by using the Compare/Move Unit)						
	47	CXi Xk	Count	Xi=(HOW MANY ONES IN Xk)	7.1	6.8	.8
move	50	SAi Aj+k	Set	Ai=Aj+k IF i>5, M(Ai)=Xi IF i>0 AND i<=5, Xi=M(Ai)	.9-1.5	.6-1.2	.3
move	*51-57 resemble 50, but replace Aj+k by: Bj+k, Xj+k, Xj+Bk, Aj+Bk, Aj-Bk, Bj+Bk, Bj-Bk)*						
move	60	SBi Aj+k	Set	IF i>0, Bi=Aj+k	.8	.5	.3
move	*60-67 resemble 50-57)*						
move	70	SXi Aj+k	Set	Xi=Aj+k	.9	.6	.3
move	*70-77 resemble 50-57; but in the UAL column, replace Xj by Xj[17 TO 0])*						

IBM 360

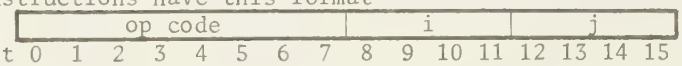
The IBM 360 family is the world's most popular family of maxicomputers. The family contains four series you can choose from: the 360 series (which is the oldest), the 370 series (which is newer), the 4300 series (which is very new and low-cost), and the 3000 series (which is very new and very fast).

The 360 series consists of many models: the 360/20, 360/30, 360/40, 360/44, 360/50, 360/65, 360/67, 360/75, 360/85, and 360/195. The 370 series consists of many models also:

MODEL	ADDING TWO 32-BIT INTEGERS	ADDING TWO 5-DIGIT BCD NUMBERS	MULTIPLYING TWO 32-BIT INTEGERS	DIVIDING TWO 32-BIT INTEGERS
370/115	14.5 μs	107.6 μs	226.1 μs	246.4 μs
370/125	9.8 μs	46.1 μs	143.4 μs	243.7 μs
370/135	4.2 μs	40.9 μs	25.5 μs	41.9 μs
370/145	2.1 μs	11.9 μs	20.1 μs	34.8 μs
370/155	1.0 μs	4.9 μs		
370/158	.8 μs	3.8 μs		
370/165	.16 μs	1.4 μs		
370/168	.13 μs			
370/195	.054 μs			

(For the lower right part of that table, data is not available.) The 4300 series consists of the 4321, 4331, and 4341. The 3000 series consists of the 3031, 3032, 3033, and 3081.

In IBM machine language, each instruction begins with an eight-bit op code. Some instructions have this format—

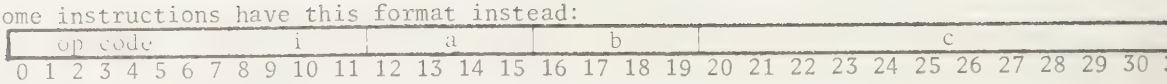


—so the leftmost eight bits form the op code, the next four bits form i, and the next four bits form j. Here are the op codes (for the IBM 370/168 operating in "BC mode"):

	OP CODE (HEXADECIMAL)	MNEMONIC	WHAT THE MNEMONIC STANDS FOR	UAL
	#04	SPM	Set Program Mask	F[4 TO 39]=Ri[2 TO 7]
branch R	#05	BALR	Branch and Link R	Ri=F[32 TO 39] P IF j>0, P=Rj
branch R	#06	BCTR	Branch on CounT R	Ri=Ri-1 IF (j>0) AND (Ri<>0), P=Rj
branch R	#07	BCR	Branch on Condition R	IF (j 0) AND (i[F[34 TO 35]]=1), P=Rj
storage key	#08	SSK	Set Storage Key	a privileged instruction
storage key	#09	ISK	Insert Storage Key	a privileged instruction
	#0A	SVC	SuperVisor Call	F[16 TO 23]=0 F[24 TO 31]=#ij M32L2=F P F P=M96L2
long strings	#0E	MVCL	MoVe Characters Long	M(Ri)L(R(i+1)[8 TO 31])=M(Rj)L(R(j+1)[8 TO 31]) PADDED BY R(j+1)[0 TO
long strings	#0F	CLCL	Compare Logical Characters Long	TEST M(Ri)L(R(i+1)[8 TO 31])-M(Rj)L(j+1)[8 TO 31] UNSIGNED PADDED BY R(j+1)[0 TO

logic R	#10 LPR	Load Positive R	IF Rj>=0, Ri=Rj IF Rj<0, Ri=-Rj
	#11 LNR	Load Negative R	IF Rj<=0, Ri=Rj IF Rj>0, Ri=-Rj
	#12 LTR	Load and Test R	Ri=Rj TEST Ri
	#13 LCR	Load Complement R	Ri=-Rj
	#14 NR	aNd R	Ri=Ri AND Rj
	#15 CLR	Compare Logical R	TEST Ri-Rj UNSIGNED
	#16 OR	Or R	Ri=Ri OR Rj
	#17 XR	eXclusive-or R	Ri=Ri XOR Rj
	#18 LR	Load R	Ri=Rj
	#19 CR	Compare R	TEST Ri-Rj
integers R	#1A AR	Add R	Ri=Ri+Rj
	#1B SR	Subtract R	Ri=Ri-Rj
	#1C MR	Multiply R	RiL2=R(i+1)*Rj
	#1D DR	Divide R	R(i+1) REM Ri=(Ri R(i+1))/Rj
	#1E ALR	Add Logical R	Ri=Ri+Rj UNSIGNED
	#1F SLR	Subtract Logical R	Ri=Ri-Rj UNSIGNED
	#20 LPDR	Load Positive Double Register	IF Fj>=0, Fi=Fj IF Fj<0, Fi=-Fj
	(#20-#23 resemble #10-#13)		
	#24 HDR	Halve Double Register	Fi=Fj/2 REAL
	#25 LRDR	Load Rounded Double Register	Fi=FjL2 ROUNDED
	#26 MXR	Multiply eXtended Register	FiL2=FiL2*FjL2 REAL
	#27 MXDR	Multiply eXtended Double Reg.	FiL2=Fi*Fj REAL
	#28 LDR	Load Double Register	Fi=Fj
	#29 CDR	Compare Double Register	TEST Fi-Fj REAL
double precision	#2A ADR	Add Double Register	Fi=Fi+Fj REAL
	#2B SDR	Subtract Double Register	Fi=Fi-Fj REAL
	#2C MDR	Multiply Double Register	Fi=Fi*Fj REAL
	#2D DDR	Divide Double Register	Fi=Fi/Fj REAL
	#2E AWR	Add W Register	Fi=Fi+Fj REAL UNNORM
	#2F SWR	Subtract W Register	Fi=Fi-Fj REAL UNNORM
	#30 LPER	Load Positive Exponent'l Reg.	IF Fj[LEFT]>=0, Fi[LEFT]=Fj[LEFT] IF Fj[LEFT]<0, Fi[LEFT]=-Fj[LEFT]
	(#31-#33 resemble #20-#25)		
	#36 AXR	Add eXtended Register	FiL2=FiL2+FjL2 REAL
	#37 SXR	Subtract eXtended Register	FiL2=FiL2-FjL2 REAL
	#38 LER	Load Exponential Register	Fi[LEFT]=Fj[LEFT]
	(#39-#3F resemble #28-#2F: replace Fi by Fi[LEFT]; replace Fj by Fj[LEFT]; in the mnemonic replace W by U; an exception is #3C, which begins "Fi=")		

Some instructions have this format instead:



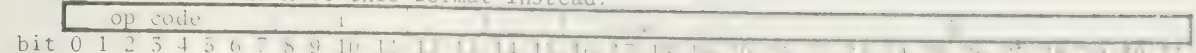
Here's how to compute j:

```
if a≠0 and b≠0, j=Ra+Rb+c
if a≠0 and b=0, j=Ra   +c
if a=0 and b≠0, j=   Rb+c
if a=0 and b=0, j=      c
```

The following op codes use that format:

	OP CODE	MNEM.	WHAT THE MNEMONIC STANDS FOR	UAL
branch	#40	STH	STore H	Hj=Ri[RIGHT]
	#41	LA	Load Address	Ri=j
	#42	STC	STore Character	Mj=Ri[24 TO 31]
	#43	IC	Insert Character	Ri[24 TO 31]=Mj
	#44	EX	EXecute	IF i=0, DO Hj IF i>0, DO Mj (M(j+1) OR Ri[24 TO 31])
	#45	BAL	Branch And Link	Ri=F[32 TO 39] P P=j
	#46	BCT	Branch on CounT	Ri=Ri-1 IF Ri<>0, P=j
	#47	BC	Branch on Condition	IF i[F[34 TO 35]]=1, P=j
H	#48	LH	Load H	Ri=Hj
	(#48-#4B resemble #18-#1B)			
	#4C	MH	Multiply H	Ri=Ri*Hj
convert BCD	#4E	CVD	ConVert to Decimal	WjL2=Ri TO DECIMAL
	#4F	CVB	ConVert to Binary	Ri=WjL2 TO BINARY
logic & integers	#50	ST	STore	Wj=Ri
	#54	N	aNd	Ri=Ri AND Wj
	(#54-#5F resemble #14-#1F)			
quadruple precision reals	#60	STD	STore Double	WjL2=Fi
	#67	MXD	Multiply eXtended Double	FiL2=Fi*WjL2 REAL
	#68	LD	Load Double	Fi=WjL2
	(#68-#6F resemble #28-#2F)			
	#70	STE	STore Exponential	Wj=Fi[LEFT]
	#78	LE	Load Exponential	Fi[LEFT]=Wj
	(#78-#7F resemble #38-#3F)			

Some instructions have this format instead:



bit 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

For such instructions, here's how to compute k:

if a≠0, k=Ra+b
if a=0, k= b

The following op codes use that format:

	OP CODE	MNEM.	WHAT THE MNEMONIC STANDS FOR	UAL
direct the I/O	#80	SSM	Set System Mask	a privileged instruction
	#82	LPSW	Load Program Status Word	F P=WkL2
	#83	none	diagnose	a privileged instruction
	#84	WRD	WRite Direct	a privileged instruction
	#85	RDD	ReaD Direct	a privileged instruction

branch on index	#86	BXH	Branch if indeX Higher	Ri=Ri+Rj
				IF (j IS ODD) AND (Ri>Rj), P=Wk
				IF (j IS EVEN) AND (Ri>R(j+1)), P=Wk

	#87	BXLE	Branch if indeX Lower or Eq.	Ri=Ri+Rj
				IF (j IS ODD) AND (Ri<=Rj), P=Wk
				IF (j IS EVEN) AND (Ri<=Rj), P=Wk

shift	#88	SRL	Shift Right Logical	SHIFTRk Ri
	#89	SLL	Shift Left Logical	SHIFTLk Ri
	#8A	SRA	Shift Right Arithmetically	SHIFTRAk Ri
	#8B	SLA	Shift Left Arithmetically	SHIFTLAk Ri
	#8C	SRDL	Shift Right Double Logical	SHIFTRk RiL2
	(#8C-#8F resemble #88-#8B)			
	#90	STM	STore Multiple	WkL(j-i+1)=RiL(j-i+1)
	#91	TM	Test Masked	TEST BITS #ij OF Mk
	#92	MVI	MoVe Immediate	Mk=#ij
	#93	TS	Test and Set	F[34]=0 F[35]=Mk[0] Mk=#FF
	#94	NI	aNd Immediate	Mk=Mk AND #ij
	(#94-#97 resemble #14-#17)			
	#98	LM	Load Multiple	RiL(j-i+1)=WkL(j-i+1)
system mask	#99	CLRIO	CLeaR I/O	a privileged instruction
	#9C	SIO (or SIOF)	Start I/O (Fast)	a privileged instruction
	#9D	TIO	Test I/O	a privileged instruction
	#9E	HIO (or HDV)	Halt I/O (or DeVice)	a privileged instruction
	#9F	TCH	Test CHannel	a privileged instruction
	#AC	STNSM	STore then aNd System Mask	a privileged instruction
	#AD	STOSM	STore then Or System Mask	a privileged instruction
	#AE	SIGP	SIGnal Processor	a privileged instruction
	#AF	MC	Monitor Call	IF MONITORMASK[j-1, Hi4$-j, F[25]-1, W15C k
	#B1	LRA	Load Real Address	a privileged instruction
	#B2	many	time, etc.	a privileged instruction
	#B6	STCTL	STore ConTroL	a privileged instruction
	#B7	LCTL	Load ConTroL	a privileged instruction
compare & swap	#BA	CS	Compare and Swap	IF Ri=Wk, Wk=Rj IF NOT, Ri=Wk
	#BB	CDS	Compare Double and Swap	IF RiL2=WkL2, WkL2=RjL2 IF NOT, RiL2=WkL2
masked characters	#BD	CLM	Compare Logical Masked TEST (BYTES j OF Ri-MkL(HOW MANY ONES IN j) UNSIGNED	
	#BE	STCM	STore Characters Masked	MkL(HOW MANY ONES IN j)=BYTES j OF Ri
	#BF	ICM	Insert Characters Masked	BYTES j OF Ri=MkL(HOW MANY ONES IN j)

Op codes #F1-#F9 handle BCD and COBOL's PICTURE.

MICROCOMPUTERS

A *microcomputer* is a low-cost computer. The CPU inside a microcomputer is called a *microprocessor*. Usually, a microprocessor is a single chip.

The first microprocessor ever invented was manufactured by Intel, which called it the "Intel 4004". Intel began shipping the 4004 to customers in June 1971. The 4004 had a 4-bit word: that is, the CPU handled only 4 bits simultaneously.

A few months later (December 1971), Intel began shipping a better CPU, called the 8008. The 8008 had an 8-bit word: the CPU handled 8 bits simultaneously.

In 1973, Intel began shipping an even better CPU, called the 8080. Like the 8008, the 8080 had an eight-bit word. The 8080 was better than the 8008 because the 8080 ran faster, contained more registers, understood more op codes, and could handle a larger RAM.

Today, you can buy several versions of the 8080. The "typical" version is called the 8080A. Faster versions are called the 8080A-1 and the 8080A-2. An even faster version is manufactured by Advanced Micro Devices (AMD), which has a license from Intel; that version is called the 9080A-4.

While Intel was inventing the 8080, Motorola was simultaneously inventing an alternative microprocessor, called the Motorola 6800. Motorola began shipping the 6800 to customers in April 1974. Like the Intel 8080, the 6800 is an improvement on the 8008.

Which is better: Intel's 8080 or Motorola's 6800? Intel's 8080 is more "powerful": it contains a greater variety of registers and a greater variety of op codes. On the other hand, Motorola's 6800 is easier to learn how to program; it's simple and elegant, like a PDP-11. It's as neat as a garden. (By comparison, the 8080 is as confusing as a jungle.)

When Microsoft wrote "8K BASIC" for both CPU's, Microsoft discovered that the BASIC ran 40% faster on the 6800 than on the 8080A, but unfortunately consumed 10% more RAM. So there was no clear "winner".

Intel's 8080 and Motorola's 6800 both became popular. The heads of those companies became rich. The hot-shot engineers who worked for those companies grew jealous of their bosses' riches and quit, to start their *own* companies.

The top engineers who left Intel formed a company called Zilog. The top engineers who left Motorola formed a company called MOS Technology.

Zilog invented a CPU called the Z-80. It resembled Intel's 8080 but was better. It soon began to outsell the 8080.

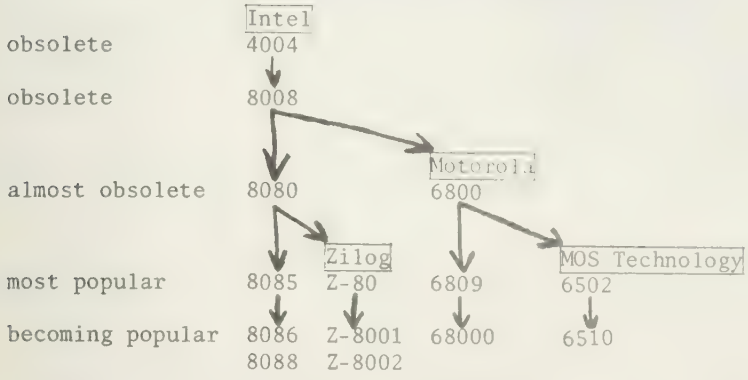
MOS Technology invented a CPU called the 6502. It resembled Motorola's 6800 but was in some ways better. It soon began to outsell the 6800.

Those new companies—Zilog and MOS Technology—needed lots of capital. (That was especially true of MOS Technology, because that company was sued by Motorola for stealing patents, and lost the suit, and had to pay Motorola $200,000 plus heavy legal fees.) So Zilog and MOS Technology both tried to convince large companies to donate money—or at least *invest* money. They succeeded: Zilog got money from Exxon, and MOS Technology got money from Commodore. But in return for that money, Zilog and MOS Technology had to give up their independence. Today, Zilog is part of Exxon, and MOS Technology is part of Commodore.

Zilog's Z-80 is the CPU in computers by Radio Shack (TRS-80 models 1, 2, 3, 4, 4P, and 12), Timex Sinclair, Coleco, and most CP/M computers (such as Kaypro, Morrow, and Cromemco). MOS Technology's 6502 is the CPU in computers by Commodore, Apple, and Atari.

When Intel and Motorola saw Zilog and MOS Technology taking over the world, Intel and Motorola retaliated, by inventing even newer and better microprocessors. To improve on Intel's own 8080, Intel invented the 8085, 8086, 8088, and 80186. To improve on Motorola's own 6800, Motorola invented the 6809 and 68000. Those newest microprocessors are being used in the newest microcomputers. For example, Intel's 8085 is used in Radio Shack's TRS-80 model 100, Intel's 8088 is used in the IBM Personal Computer, Intel's 80186 is used in the Tandy 2000, Motorola's 6809 is used in the Radio Shack Color Computer, and Motorola's 68000 is used in Radio Shack's TRS-80 model 16 and Apple's Macintosh. So Intel and Motorola might have the last laugh.

This family tree summarizes what's been going on:



Here's the speed of some famous CPU's:

MODEL	HOW LONG A "MACHINE CYCLE" IS	HOW MANY "MACHINE CYCLES" YOU NEED, TO ADD TOGETHER TWO 8-BIT INTEGERS	TOTAL TIME, TO ADD TOGETHER TWO 8-BIT INTEGERS
6800	1 µs	2	2 µs
68A00	.65 µs	2	1.3 µs
68B00	.5 µs	2	1 µs
6502	.5 µs	2	1 µs
6502C	.25 µs	2	.5 µs
8080A	.48 µs	4	1.92 µs
8080A-2	.375 µs	4	1.5 µs
8080A-1	.325 µs	4	1.3 µs
8085	.325 µs	4	1.3 µs
9080A-4	.25 µs	4	1 µs
Z-80	.4 µs	4	1.6 µs
Z-80A	.25 µs	4	1 µs
Z-80B	.165 µs	4	.66 µs

Notice that adding together two short integers takes 2 or 4 cycles, depending on your CPU.

Besides adding, the CPU can do other operations, shown in the tables on the next few pages. Throughout the tables, "i" means any octal digit (from 0 to 7), "j" means any pair of hexadecimal digits (from #00 to #FF), and "k" means any four hexadecimal digits (from #0000 to #FFFF). For example, if the table says the CPU can do Mi=Mi+1, it means you can make the CPU do each of these: M0=M0+1, M1=M1+1, M2=M2+1, M3=M3+1, M4=M4+1, M5=M5+1, M6=M6+1, and M7=M7+1.

Throughout the tables, "c" means any of these IF-clauses....

IF-CLAUSE, IN UAL:	IF N=1	IF N=0	IF Z=1	IF Z=0	IF C=1	IF C=0	IF V=1	IF V=0
MNEMONIC ON 6800 & 6502:	MI	PL	EQ	NE	CS	CC	VS	VC
MNEMONIC ON 8080 & Z-80:	M	P	Z	NZ	C	NC	PE	PO

On the 8080, which doesn't have a V flag, the mnemonic PE means "IF PARITY=1", and the mnemonic PO means "IF PARITY=0". On the 6800, additional mnemonics are available:

UAL:	IF (Z OR C)=1	if =0	IF (N XOR V)=1	if =0	IF (Z OR (N XOR V))=1	if =0
6800:	LS	HI	LT	GE	LE	GT

I'll use these symbols:

SYMBOL	MEANING
r	A *or* B
m	Mj *or* Mk *or* M(j+X)
mj	Mj *or* Mk *or* M(j+X) or j
mm	MjL2 *or* MkL2 *or* M(j+X)L2
mmk	MjL2 *or* MkL2 *or* M(j+X)L2 *or* k
mr	Mk *or* M(j+X) *or* A *or* B
xk	j+X *or* k

Here's the instruction set:

	PURPOSE	UAL	MNEMONIC	CYCLES
math	increment	mr=mr+1	INC	2-7
		S=S+1	INS	4
		X=X+1	INX	4
	decrement	mr=mr-1	DEC	2-7
		S=S-1	DES	4
		X=X-1	DEX	4
	add	r=r+mj	ADD	2-5
		r=r+mj+C	ADC	2-5
		A=A+B	ABA	2
	subtract	r=r-mj	SUB	2-5
		r=r-mj-C	SBC	2-5
		A=A-B	SBA	2
	negate	mr=-mr	NEG	2-7
	adjust	ADJUST A	DAA	2
logic	and	r=r AND mj	AND	2-5
	or	r=r OR mj	ORA	2-5
	xor	r=r XOR mj	EOR	2-5
	not	mr=NOT mr	COM	2-7
	shiftl	SHIFTL mr	ASL	2-7
	shiftr	SHIFTR mr	LSR	2-7
		SHIFTRA mr	ASR	2-7
	rotatel	ROTATEL C mr	ROL	2-7
	rotater	ROTATER C mr	ROR	2-7
	clear	mr=0	CLR	2-7
		V=0	CLV	2
		C=0	CLC	2
		PRIORITY=0	CLI	2
	set	V=1	SEV	2
		C=1	SEC	2
		PRIORITY=1	SEI	2
	test	TEST mr	TST	2-7
		TEST r-mj	CMP	2-5
		TEST X-mmk	CPX	3-6
		TEST A-B	CBA	2
		TEST r AND mj	BIT	2-5

	PURPOSE	UAL	MNEMONIC	CYCLES
moving bytes	transfer	A=B	TBA	2
		B=A	TAB	2
		A=F	TAP	2
		F=A	TPA	2
		S=X-1	TXS	4
		X=S+1	TSX	4
	load	r=mj	LDA	2-5
		S=mmk	LDS	2-6
		X=mmk	LDX	3-6
	store	m=r	STA	4-6
		mm=S	STS	5-7
		mm=X	STX	5-7
	push	M(S)=r, S=S-1	PSH	4
	pull	S=S+1, r=M(S)	PUL	4
program counter	jump	P=xk	JMP	3-4
		P=P+j	BRA	4
	jump if	IF c, P=P+j	Bc	4
	continue	CONTINUE	NOP	2
	wait	M(S-1)L2=P, M(S-3)L2=X, M(S-4)=A, M(S-5)=B, M(S-6)=F, S=S-7, WAIT	WAI	9-∞
	call	M(S-1)L2=P, S=S-2, P=xk	JSR	8-9
		M(S-1)L2=P, S=S-2, P=P+j	BSR	8
		M(S-1)L2=P, M(S-3)L2=X, M(S-4)=A, M(S-5)=B, M(S-6)=F, S=S-7, P=M#FFFAL2	SWI	12
	return	S=S+2, P=M(S-1)L2	RTS	5
		S=S+7, F=M(S-6), B=M(S-5), A=M(S-4), X=M(S-3)L2, P=M(S-1)L2	RTI	10

Instruction set for MOS Technology's 6502

I'll use these symbols:

SYMBOL	MEANING
m	Mj *or* Mk
mj	Mj *or* Mk *or* j
mx	Mj *or* Mk *or* M(j+X)
mxk	Mj *or* Mk *or* M(j+X) *or* M(k+X)
mxkj	Mj *or* Mk *or* M(j+X) *or* M(k+X) *or* j
mxka	Mj *or* Mk *or* M(j+X) *or* M(k+X) *or* A
my	Mj *or* Mk *or* M(j+Y)
mykj	Mj *or* Mk *or* M(j+Y) *or* M(k+Y) *or* j
mykj	Mj *or* Mk *or* M(j+Y) *or* M(k+Y) *or* j
mxy	Mj *or* Mk *or* M(j+X) *or* M(k+X) *or* M(M(j+X)L2) *or* M(k+Y) *or* M(MjL2+Y)
mxyj	Mj *or* Mk *or* M(j+X) *or* M(k+X) *or* M(M(j+X)L2) *or* M(k+X) *or* M(MjL2+Y) *or* j
mk	MkL2 *or* k

Here's the instruction set; for mnemonics ADC and SBC, the arithmetic is decimal (instead of binary) if the flag register's DECIMAL bit is 1:

	PURPOSE	UAL	MNEMONIC	CYCLES
math	increment	mxk=mxk+1	INC	5-7
		X=X+1	INX	2
		Y=Y+1	INY	2
	decrement	mxk=mxk-1	DEC	5-7
		X=X-1	DEX	2
		Y=Y-1	DEY	2
	add	A=A+mxyj+C	ADC	2-6
	subtract	A=A-mxyj-C	SBC	2-6
logic	and	A=A AND mxyj	AND	2-6
	or	A=A OR mxyj	ORA	2-6
	xor	A=A XOR mxyj	EOR	2-6
	shiftl	SHIFTL mxka	ASL	2-7
	shiftr	SHIFTR mxka	LSR	2-7
	rotatel	ROTATEL C mxka	ROL	2-7
	rotater	ROTATER C mxka	ROR	2-7
	clear	V=0	CLV	2
		C=0	CLC	2
		DECIMAL=0	CLD	2
		PRIORITY=0	CLI	2
	set	C=1	SEC	2
		DECIMAL=1	SED	2
		PRIORITY=1	SEI	2
	test	TEST A-mxyj	CMP	2-6
		TEST X-mj	CPX	2-4
		TEST Y-mj	CPY	2-4
		TEST A AND m	BIT	3-4

	PURPOSE	UAL	MNEMONIC	CYCLES
moving bytes	transfer	A=X	TXA	2
		X=A	TAX	2
		A=Y	TYA	2
		Y=A	TAY	2
		S=X	TXS	2
		X=S	TSX	2
	load	A=mxyj	LDA	2-6
		X=mykj	LDX	2-4
		Y=mxkj	LDY	2-4
	store	mxy=A	STA	5-6
		my=X	STX	3-4
		mx=Y	STY	3-4
	push	M(S)=A, S=S-1	PHA	3
		M(S)=F, S=S-1	PHP	3
	pull	S=S+1, A=M(S)	PLA	4
		S=S+1, F=M(S)	PLP	4
program counter	jump	P=mk	JMP	3-5
	jump if	IF c, P=P+j	Bc	2
	continue	CONTINUE	NOP	2
	call	M(S-1)L2=P, S=S-2, P=k	JSR	6
		M(S-1)L2=P, M(S-2)=F, S=S-3, PRIORITY=1, BREAK=1, P=M#FFFEL2	BRK	7
	return	S=S+2, P=M(S-1)L2, P=P+1	RTS	6
		S=S+3, M(S-2)=F, M(S-1)L2=P, P=P+1	RTI	6

Instruction set for Intel's 8080

I'll use these symbols:

SYMBOL	MEANING
r	A *or* B *or* CR *or* D *or* E *or* H *or* L
rm	A *or* B *or* CR *or* D *or* E *or* H *or* L *or* M(H L)
rs	B CR *or* D E *or* H L *or* S
raf	B CR *or* D E *or* H L *or* A F
m	M(B CR) *or* M(D E)

Here's the instruction set:

	PURPOSE	UAL	MNEMONIC	CYCLES
math	increment	rm=rm+1	INR	5-10
		rs=rs+1	INX	5
	decrement	rm=rm-1	DCR	5-10
		rs=rs-1	DCX	5
	add	A=A+rm	ADD	4-7
		A=A+rm+C	ADC	4-7
		A=A+j	ADI	7
		A=A+j+C	ACI	7
		H L=(H L)+rs	DAD	10
	subtract	A=A-rm	SUB	4-7
		A=A-rm-C	SBB	4-7
		A=A-j	SUI	7
		A=A-j-C	SBI	7
	adjust	ADJUST A	DAA	4
logic	and	A=A AND rm	ANA	4-7
		A=A AND j	ANI	7
	or	A=A OR rm	ORA	4-7
		A=A OR j	ORI	7
	xor	A=A XOR rm	XRA	4-7
		A=A XOR j	XRI	7
	not	A=NOT A	CMA	4
		C=NOT C	CMC	4
	rotatel	ROTATEL A	RLC	4
		ROTATEL C A	RAL	4
	rotater	ROTATER A	RRC	4
		ROTATER C A	RAR	4
	clear	PRIORITY=0	EI	4
	set	PRIORITY=1	DI	4
		C=1	STC	4
	test	TEST A-rm	CMP	4-7
		TEST A-j	CPI	7

	PURPOSE	UAL	MNEMONIC	CYCLES
moving bytes	transfer	$rm_1=rm_2$	MOV	5-7
	load	A=Mk	LDA	13
		A=m	LDAX	7
		H L=M(k+1) Mk	LHLD	16
		rs=k	LXI	10
		rm=j	MVI	7
	store	Mk=A	STA	13
		m=A	STAX	7
	exchange	H L= =D E	XCHG	4
		H L= =S	SPHL	5
		H L= =M(S+1) M(S)	XTHL	18
	input	A=PORTj	IN	10
	output	PORTj=A	OUT	10
	push	S=S-2, M(S+1) M(S)=raf	PUSH	11
	pull	raf=M(S+1) M(S), S=S+2	POP	10
program counter	jump	P=k	JMP	10
		P=H L	PCHL	5
	jump if	IF c, P=k	Jc	10
	continue	CONTINUE	NOP	4
	wait	WAIT	HLT	4-∞
	call	S=S-2, M(S+1) M(S)=P, P=k	CALL	17
		S=S-2, M(S+1) M(S)=P, P=#8*i	RST	11
	call if	IF c, S=S-2, M(S+1) M(S)=P, P=k	Cc	11-17
	return	P=M(S+1) M(S), S=S+2	RET	10
	return if	IF c, P=M(S+1) M(S), S=S+2	Rc	5-11

Instruction set for Zilog's Z-80

I'll use these symbols:

SYMBOL	MEANING
r	A *or* B *or* CR *or* D *or* E *or* H *or* L
rm	r *or* M(H L) *or* M(j+X) *or* M(j+Y)
rmj	rm *or* j
rms	rm *or* B CR *or* D E *or* H L *or* S *or* X *or* Y
rmsz	rms *or* M(B CR) *or* M(D E) *or* M(k+1) Mk *or* IV *or* RE
n	0 *or* 1 *or* 2

SY.	MEANING
am	A F *or* D E *or* M(S+1) M(S)
ax	A F *or* B CR *or* D E *or* H L *or* X *or* Y
ah	A *or* H L
ahx	A *or* H L *or* X *or* Y
u	*depends on other operands*

Here's the instruction set:

	PURPOSE	UAL	MNEMONIC	CYCLES
math	increment	rms=rms+1	INC	4-23
	decrement	rms=rms-1	DEC	4-23
		B=B-1, IF B<>0, P=P+j	DJNZ	8-13
	add	ahx=ahx+u	ADD	4-19
		ah=ah+u+C	ADC	4-19
	subtract	A=A-rmj	SUB	4-19
		ah=ah-u-C	SBC	4-19
	negate	A=-A	NEG	8
	adjust	ADJUST A	DAA	4
logic	and	A=A AND rmj	AND	4-19
	or	A=A OR rmj	OR	4-19
	xor	A=A XOR rmj	XOR	4-19
	not	A=NOT A	CPL	4
		C=NOT C	CCF	4
	shiftl	SHIFTL rm	SLA	8-23
	shiftr	SHIFTR rm	SRL	8-23
	rotatel	ROTATEL rm	RLC	8-23
		ROTATEL A	RLCA	4
		ROTATEL C rm	RL	8-23
		ROTATEL C A	RLA	4
		ROTATEL4 A[RIGHT] M(H L)	RLD	18
	rotater	*like rotatel, but change "ROTATEL" to "ROTATER"*	RRC, etc.	
	clear	PRIORITY=0	EI	4
		rm[i]=0	RES	8-23
	set	PRIORITY=1	DI	4
		C=1	SCF	4
		rm[i]=1	SET	8-23
	test	TEST A-rmj	CP	4-19
		TEST A-M(H L), H L=(H L)+1, B CR=(B CR)-1	CPI	16
		WHILE B CR<>0 AND A<>M(H L), H L=(H L)+1, B CR=(B CR)-1	CPIR	16-∞
		like the two above, but change "+" to "-"	CPD, CPDR	
		TEST rm[i]	BIT	8-20

	PURPOSE	UAL	MNEMONIC	CYCLES
moving bytes	move	rmsz=u	LD	4-20
		M(D E)=M(H L), D E=(D E)+1, H L=(H L)+1, B CR=(B CR)-1	LDI	16
		WHILE B CR<>0, M(D E)=M(H L), D E=(D E)+1, H L=(H L)+1, B CR=(B CR)-1	LDIR	16-∞
		like the two above, but change each "+1" to "-1"	LDD, LDDR	
		IMODE=n	IM	8
	exchange	am= =u	EX	4-23
		B CR D E H L= = B2 CR2 D2 E2 H2 L2	EXX	4
	input	r=PORT(u)	IN	10-11
		M(H L)=PORT(CR), H L=(H L)+1, B=B-1	INI	15
		WHILE B<>0, M(H L)=PORT(CR), H L=(H L)+1, B=B-1	INIR	15-∞
		like the two above, but change "+1" to "-1"	IND, INDR	
	output	*like input, but put "PORT(u)" or "PORT(CR)" left of the equal sign*	OUT, etc.	
	push	S=S-2, M(S+1) M(S)=ax	PUSH	11-15
	pull	ax=M(S+1) M(S), S=S+2	POP	10-14
program counter	jump	P=hxk	JP	4-10
		P=P+j	JR	12
	jump if	IF c, P=k	JP c	10
		IF Z=1, P=P+j	JR Z	7-12
		IF Z=0, P=P+j	JR NZ	7-12
		IF C=1, P=P+j	JR C	7-12
		IF C=0, P=P+j	JR NC	7-12
	continue	CONTINUE	NOP	4
	wait	WAIT	HALT	4-∞
	call	S=S-2, M(S+1) M(S)=P, P=k	CALL	17
		S=S-2, M(S+1) M(S)=P, P=#8*i	RST	11
	call if	IF c, S=S-2, M(S+1) M(S)=P, P=k	CALL c	10-17
	return	P=M(S+1) M(S), S=S+2	RET	10
		PRIORITY=0, IRETURN=2, P=M(S+1) M(S), S=S+2	RETI	14
		PRIORITY=0, IRETURN=3, P=M(S+1) M(S), S=S+2	RETN	14
	return if	IF c, P=M(S+1) M(S), S=S+2	RET c	5-11

CHAPTER 9

ABSTRACT APPLICATIONS

3-D DRAWING

COORDINATES

Try this experiment. Put your finger on the bridge of your nose (between your eyes). Now move your finger 2 inches to the right (so that your finger is close to your right eye). Then move your finger 3 inches up (so that your finger is near the upper right corner of your forehead). From there, move your finger 8 inches forward (so that your finger is 8 inches in front of your forehead).

Your finger's current position is called (2,3,8), because you reached it by moving 2 inches right, then 3 inches up, then 8 inches forward. The 2 is called the *X coordinate*; the 3 is called the *Y coordinate*; the 8 is called the *Z coordinate*.

You can reach any point in the universe by the same method! Start at the bridge of your nose, and get to the point by moving right (or left), then up (or down), then forward (or back).

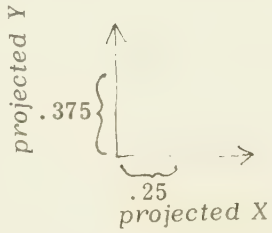
The distance you move to the right is called the *X coordinate* (if you move to the left instead, the X coordinate is a negative number). The distance you move up is called the *Y coordinate* (if you move down instead, the Y coordinate is a negative number). The distance you move forward is called the *Z coordinate* (if you move back instead, the Z coordinate is a negative number).

DRAWING AN OBJECT

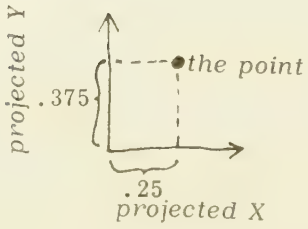
To draw a picture of a three-dimensional object, put the object in front of you, and then follow these instructions....

Pick a point on the object. (If the object has corners, pick one of the corners.) Figure out that point's X, Y, and Z coordinates (by putting your finger on the bridge of your nose and then seeing how far you must move your finger right, up, and foward to reach the object).

Then compute the point's *projected X coordinate* (which is X/Z) and the point's *projected Y coordinate* (which is Y/Z). For example, if X is 2 and Y is 3 and Z is 8, the projected X coordinate is 2/8 (which is .25) and the projected Y coordinate is 3/8 (which is .375). On graph paper, plot the projected X coordinate and the projected Y coordinate, like this:



Then plot the point:



Go through the same procedure for every point on the object (or at least on the corners). Connect the dots and— presto!— you have a three-dimensional picture of the object! And the picture is mathematically accurate! It's what artists call a "true perspective drawing".

To make the picture look traditionally beautiful, place the object slightly to the left of you and slightly below your eye level, so that all the X and Y coordinates become negative.

COMPUTERIZING THE PROCESS

You can program the computer so that if you input a point's X coordinate, Y coordinate, and Z coordinate, the computer will calculate the projected X coordinate (from dividing X by Z) and the projected Y coordinate (from dividing Y by Z) and plot the point on the computer's screen (by using high-resolution graphics).

The easiest way to draw three-dimensional pictures is to buy a special three-dimensional arm that attaches to an Apple computer. To draw a picture of an object, move the mechanical arm until the arm's finger touches the object. Immediately the arm's software computes the X coordinate, Y coordinate, and Z coordinate of the touched point. You don't need a ruler! It also computes the projected X coordinate and the projected Y coordinate and plots the point on your television. If you have a graphics printer, the software also plots the point on your printer's paper.

NUMERICAL ANALYSIS

TWIN PROBLEMS THAT CONFUSE THE COMPUTER

Look at this pair of problems:

Problem 1. Solve $x^2-4x+4=0$.
Problem 2. Solve $x^2-4x+3.9999999=0$.

Those two problems resemble each other. The only difference is that where the second problem says 3.9999999, the first problem says 4. But the correct solutions to the problem are quite different from each other:

The correct solution to problem 1 is "x = 2".
The correct solution to problem 2 is "x = 1.9996838 or x = 2.0003162".

Unfortunately, problem 1 looks so similar to problem 2 that the computer can hardly tell the difference. On some computers, 3.9999999 is treated as if it were 4. If you ask such a computer to solve problem 2, it will solve problem 1 instead, and its answer to problem 2 will therefore be wrong.

Here's another pair of problems:

Problem 3. Solve the equation $(x-1)*(x-2)*(x-3)*(x-4)*...*(x-20)=0$.

Problem 4. Solve the equation $(x-1)*(x-2)*(x-3)*(x-4)*...*(x-20)-\frac{1}{2^{23}}\cdot x^{19}=0$.

Those two problems resemble each other. In fact, if you "expand the polynomial", you'll see that problem 3 can be rewritten like this:

$x^{20}-210x^{19}+...=0$.

Problem 4 can be rewritten like this:

$x^{20}-(210+\frac{1}{2^{23}})x^{19}+...=0$.

So the only difference between problem 3 and problem 4 is that problem 4 says $210+\frac{1}{2^{23}}$ instead of 210. Because $\frac{1}{2^{23}}$ is such a small fraction, some computers can't tell the difference between 210 and $210+\frac{1}{2^{23}}$, and therefore can't distinguish problem 4 from problem 3, and therefore say the same answer to problem 4 as to problem 3. But the *correct* solutions to the two problems are different:

The correct solution to problem 3 is "x=1 or x=2 or x=3 or x=4 or ... or x=20".
The correct solution to problem 4 is "x=1.000 or x=2.000 or x=3.000 or x=4.000 or x=5.000 or x=6.000 or x=7.000 or x=8.007 or x=8.917 or x=10.095±.644i or x=11.794±1.652i or x=13.992±1.519i or x=16.731±2.813i or x=19.502±1.940i or x=20.847". (i denotes $\sqrt{-1}$.)

Here's another case of "confusing twin problems". Let M be the matrix $\begin{pmatrix} 9999 & 9998 \\ 10000 & 9999 \end{pmatrix}$.

Problem 5. Let X be $\begin{pmatrix} 9999 & -9998 \\ -10000 & 9999 \end{pmatrix}$. Compute M*X and X*M.

Problem 6. Let X be $\begin{pmatrix} 9999.9999 & -9997.0001 \\ -10001 & 9998 \end{pmatrix}$. Compute M*X and X*M.

To compute M*X and X*M, you're supposed to use the definition of "matrix multiplication" as defined by college math courses. Here are the correct solutions:

The correct solution to problem 5 is "M*X=$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, and X*M=$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ also".

The correct solution to problem 6 is "M*X=$\begin{pmatrix} 1.001 & .001 \\ 0 & 1 \end{pmatrix}$, and X*M=$\begin{pmatrix} 19998.0001 & 19995.00 \\ -19999 & -19994 \end{pmatrix}$

That's another example of how a small change in a problem can make a big difference in the solution.

Here's one more case:

Problem 7. Solve the simultaneous equations "913x+659y=254 and 780x+563y=217".
Problem 8. Solve the simultaneous equations "913x+659y=254 and 780x+563y=216.999".

Although the problems are almost identical (so that some computers can't distinguish them from each other), the correct solutions are quite different:

The correct solution to problem 7 is "x=1 and y=-1".
The correct solution to problem 8 is "x=.341 and y=-.087".

That last case of "confusing twins" leads to this fascinating paradox... Suppose you tell your friendly computer to "guess" an x and y to solve problem 7:

913x+659y = 254
780x+563y = 217

If the computer guesses "x = .999 and y = -1.001", it finds:

913x+659y = 252.428
780x+563y = 215.757

But if the computer guesses "x = .341 and y = -.087" instead, it finds:

913x+659y = 254
780x+563y = 216.999, which is about 217

The computer therefore deduces that its second guess is almost perfect and much better than its first guess. But— here's the paradox— the second guess is *not* better than the first guess; the *first* guess is better, because the correct solution is "x = 1 and y = -1", which is closer to the *first* guess.

HOW TO REDUCE ROUND-OFF ERROR

By the laws of mathematics, 5+.14+.14+.14 should be the same as .14+.14+.14+5. But a calculator that holds only two significant digits gives different answers:

5+.14+.14+.14	.14+.14+.14+5
5.0	.14
+.14	+.14
5.1	.28
+.14	+.14
5.2	.42
+.14	+5.0
5.3	5.4

The correct answer is 5.42, so the calculation on the right seems more accurate. The general rule is: when adding a list of numbers, you'll get more accuracy if you begin with the numbers closest to 0. The rule is valid even on a top-quality computer, although the computer's inaccuracies are not so obvious.

By the laws of mathematics, $(3\frac{1}{5} - 3\frac{1}{7})*70$ is 4:

$$3\frac{1}{5} = 3\frac{7}{35}$$

$$-3\frac{1}{7} = 3\frac{5}{35}$$

$$\frac{2}{35} * 70 = 4$$

But if we try that calculation on our two-digit calculator, we get a totally different answer:

$$3\frac{1}{5} = 3+.2 = 3.2$$

$$-3\frac{1}{7} = 3+.14 = 3.1$$

$$.1 * 70 = 7.0$$

The general warning is: when you subtract two numbers that are almost equal (like $3\frac{1}{5}$ and $3\frac{1}{7}$), few of the digits in your answer will be correct. The warning is valid even on a top-quality computer, although the computer's inaccuracies are not so obvious.

To solve the quadratic equation $ax^2+bx+c=0$, high-school teachers say to set d equal to b^2-4ac and then use these formulas:

$$x_1 = \frac{-b+\sqrt{d}}{2a}$$

$$x_2 = \frac{-b-\sqrt{d}}{2a}$$

Unfortunately, if $\sqrt{d}$ is approximately b, the numerator of the formula for x_1 involves the subtraction of two numbers that are almost equal, and therefore generates round-off error. Use this formula instead:

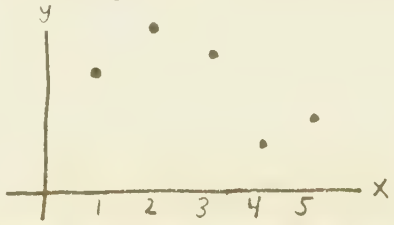
$$x_1 = \frac{2c}{-b-\sqrt{d}}$$

If $\sqrt{d}$ is approximately -b, the new formula for x_1 has a bad denominator, and the original formula is better; but since the formula for x_2 has a bad numerator now, use this formula instead:

$$x_2 = \frac{2c}{-b+\sqrt{d}}$$

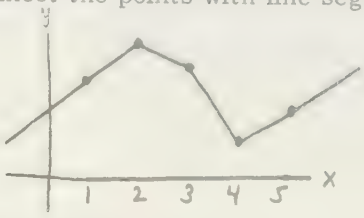
HOW TO ESTIMATE AN UNKNOWN ANSWER

Suppose you're given the value of y when x is 1, 2, 3, 4, and 5:



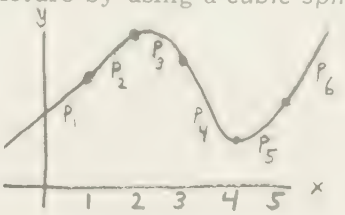
Suppose you'd like to make an "intelligent guess" as to the value of y when x is 1.5, 3.01 100, -400, and other values. Guessing y for an in-between x (such as 1.5 and 3.01) is called *interpolation*; guessing y for a very large x or a very small x (such as 100 and -40 is called *extrapolation*.

One way to guess is to connect the points with line segments:



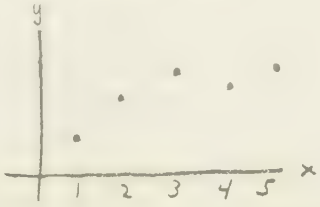
That's called *piecewise linear* estimation.

You get a much smoother picture by using a *cubic spline*:

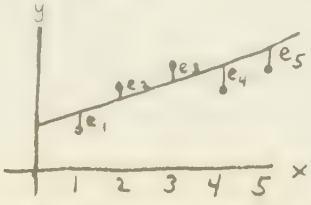


P_1 and P_6 are parts of straight lines. P_2, P_3, P_4, and P_5 are parts of four different cubics (third-degree polynomials), chosen so that, at the point where P_i meets P_{i+1}, P_i has the same *derivative* and *second derivative* as P_{i+1}. (The term "derivative" is defined by calculus.)

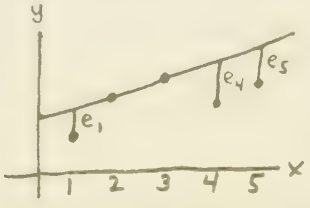
Suppose you're given these *approximate* values of y:



To estimate y for other values of x, you could use piecewise linear estimation or a cubic spline. But notice the points lie almost on a straight line:



The points aren't exactly on that line: the errors in the y values are e_1, e_2, e_3, e_4, and e_5. The line's *squared error* is $e_1^2 + e_2^2 + e_3^2 + e_4^2 + e_5^2$. This line has a larger squared error:



The line having the smallest squared error is called the *least-square line*, and is considered to be the line that best approximates the data.

HOW TO SOLVE AN EQUATION

Suppose you want to solve a tough equation, such as $2^x+x^3=20$. Rewrite it to make the right side be 0:

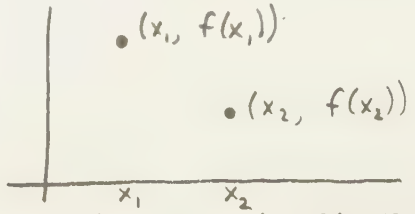
$2^x+x^3-20=0$

Let f(x) denote the left side of the equation:

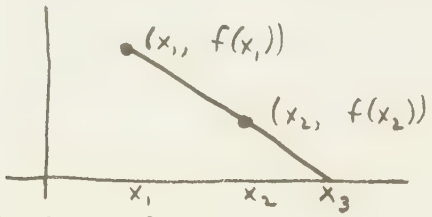
f(x) is 2^x+x^3-20

So the equation you want to solve is f(x)=0. Here's a way to solve it— called the *secant method*....

Let x_1 and x_2 be your favorite numbers. Graph $f(x_1)$ and $f(x_2)$:



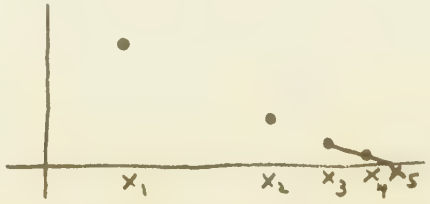
Let x_3 be where the line connecting those points hits the x axis:



Graph $f(x_3)$; it's probably close to 0:

Connect the x_2 point to the x_3 point, to find x_4:

Connect the x_3 point to the x_4 point, to find x_5:

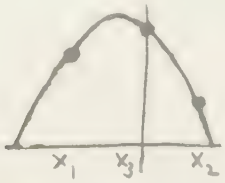


Probably you'll get closer and closer to the point where f(x) is 0.

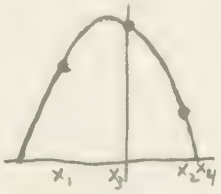
A different method, called *Muller's method*, uses parabolas instead of straight lines....

Let x_1, x_2, and x_3 be your three favorite numbers. A common choice is -.5, .5, and 0. Graph $f(x_1)$, $f(x_2)$, and $f(x_3)$:

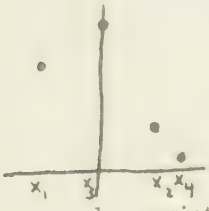
Draw the parabola that passes through those points:



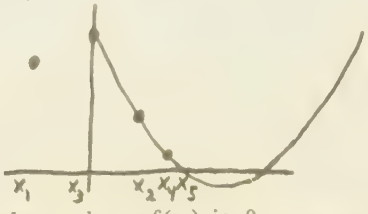
The parabola hits the x axis at two points (although the points might be what mathematicians call "imaginary"). Let x_4 be at one of those points. For best results, choose the point that's closer to the origin:



Graph $f(x_4)$:



Draw a parabola through the x_2, x_3, and x_4 points, to find x_5:



Probably you'll get closer to the place where f(x) is 0.

Usually, Muller's method moves toward the solution more rapidly than the secant method. Moreover, it can even find complex, "non real" solutions, since the x value where the parabola hits the x axis might be of the form a+bi, where i denotes $\sqrt{-1}$. (If you're interested in only solutions that are real, pretend b is 0.)

Using either the secant method or Muller's, suppose you finally find a solution of the equation $2^x+x^3-20=0$. Let s_1 denote that solution. To hunt for additional solutions, try solving the equation $\frac{2^x+x^3-20}{x-s_1} = 0$. If you find a solution of that new equation, call it s_2, and solve the equation $\frac{2^x+x^3-20}{(x-s_1)(x-s_2)} = 0$. If that produces a solution s_3, solve $\frac{2^x+x^3-20}{(x-s_1)(x-s_2)(x-s_3)} = 0$, to find s_4. The solutions s_1, s_2, s_3, s_4, etc., are all solutions of the original equation.

The round-off error will be less if $|s_1| < |s_2| < |s_3| < |s_4| < \ldots$. So when you hunt for solutions, begin by trying to find the solutions closest to zero. That's why, when Muller's parabola hits the x axis in two points, you should pick the point that's closer to zero. And that's why a common choice for x_1, x_2, and x_3 is numbers near zero.

HOW TO SOLVE SIMULTANEOUS EQUATIONS

Try to solve these simultaneous linear equations:

$$\begin{aligned} 8x + y - z &= 8 \\ 2x + y + 9z &= 12 \\ x - 7y + 2z &= -4 \end{aligned}$$

An obvious way is *Gauss-Jordan elimination*. Eliminate the first coefficient:

$$\begin{aligned} \square x + \tfrac{1}{8}y - \tfrac{1}{8}z &= 1 \quad \textit{(I multiplied by } \tfrac{1}{8}\textit{)} \\ 2x + y + 9z &= 12 \\ x - 7y + 2z &= -4 \end{aligned}$$

Eliminate everything below it:

$$\begin{aligned} x + \tfrac{1}{8}y - \tfrac{1}{8}z &= 1 \\ \tfrac{3}{4}y + \tfrac{37}{4}z &= 10 \quad \textit{(I subtracted twice the first row)} \\ -\tfrac{57}{8}y + \tfrac{17}{8}z &= -5 \quad \textit{(I subtracted the first row)} \end{aligned}$$

Eliminate the first coefficient in the second row:

$$\begin{aligned} x + \tfrac{1}{8}y - \tfrac{1}{8}z &= 1 \\ \square y + \tfrac{37}{3}z &= \tfrac{40}{3} \quad \textit{(I multiplied by } \tfrac{4}{3}\textit{)} \\ -\tfrac{57}{8}y + \tfrac{17}{8}z &= -5 \end{aligned}$$

Eliminate everything above and below it:

$$\begin{aligned} x \square - \tfrac{5}{3}z &= -\tfrac{2}{3} \quad \textit{(I subtracted } \tfrac{1}{8} \textit{ times the second row)} \\ y + \tfrac{37}{3}z &= \tfrac{40}{3} \\ \square 90z &= 90 \quad \textit{(I added } \tfrac{57}{8} \textit{ times the second row)} \end{aligned}$$

Eliminate the first coefficient in the third row:

$$\begin{aligned} x - \tfrac{5}{3}z &= -\tfrac{2}{3} \\ y + \tfrac{37}{3}z &= \tfrac{40}{3} \\ \square z &= 1 \quad \textit{(I multiplied by } \tfrac{1}{90}\textit{)} \end{aligned}$$

Eliminate everything above it:

$$\begin{aligned} x \square &= 1 \quad \textit{(I added } \tfrac{5}{3} \textit{ times the third row)} \\ y \square &= 1 \quad \textit{(I subtracted } \tfrac{37}{3} \textit{ times the third row)} \\ z &= 1 \end{aligned}$$

That's the solution. (It's depressing that so much computation was needed, to get such a simple solution!)

In that example, I pivoted on the first coefficient of the first equation, then the first coefficient of the second equation, and finally the first coefficient of the third equation. To reduce the computer's round-off error, it's better at each stage to pivot on the coefficient that has the largest absolute value, relative to the coefficients in its row. For example, at this stage—

$$\begin{aligned} x + \tfrac{1}{8}y - \tfrac{1}{8}z &= 1 \\ \tfrac{3}{4}y + \tfrac{37}{4}z &= 10 \\ -\tfrac{57}{8}y + \tfrac{17}{8}z &= -5 \end{aligned}$$

— it would've been better to pivot on $-\frac{57}{8}$ than on $\frac{3}{4}$, since $\frac{57}{8}$ divided by $\frac{47}{8}$ is bigger than $\frac{3}{4}$ divided by $\frac{37}{4}$.

Now let's go back to the original equations—

$$\begin{aligned} 8x + y - z &= 8 \\ 2x + y + 9z &= 12 \\ x - 7y + 2z &= -4 \end{aligned}$$

— and solve them by a different method, called *Gauss-Seidel iteration*. In each equation, find the variable whose coefficient has the largest absolute value, and put that variable on one side:

$$x = -\frac{1}{8}y + \frac{1}{8}z + 1$$

$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3}$$

$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7}$$

Begin by guessing that x, y, and z are all 0. You'll improve the guesses, by using those equations. Here are the calculations, to three decimal places:

$$x = 0$$
$$y = 0$$
$$z = 0$$
$$x = -\frac{1}{8}y + \frac{1}{8}z + 1 = 1.000$$
$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7} = .714$$
$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3} = 1.032$$
$$x = -\frac{1}{8}y + \frac{1}{8}z + 1 = 1.041$$
$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7} = 1.014$$
$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3} = .990$$
$$x = -\frac{1}{8}y + \frac{1}{8}z + 1 = .997$$
$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7} = .996$$
$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3} = 1.002$$
$$x = -\frac{1}{8}y + \frac{1}{8}z + 1 = 1.001$$
$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7} = 1.000$$
$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3} = 1.000$$
$$x = -\frac{1}{8}y + \frac{1}{8}z + 1 = 1.000$$
$$y = \frac{1}{7}x + \frac{2}{7}z + \frac{4}{7} = 1.000$$
$$z = -\frac{2}{9}x - \frac{1}{9}y + \frac{4}{3} = 1.000$$

In the example, x and y and z gradually approach the correct solutions.

Is Gauss-Seidel iteration better or worse than Gauss-Jordan elimination?

A disadvantage of Gauss-Seidel iteration is: it doesn't guarantee you'll approach the correct solution. If you follow the advice about pivoting on the coefficient that has the largest relative absolute value, you stand a better chance of approaching the correct solution, but there's no guarantee.

On the other hand, if Gauss-Seidel iteration *does* approach a solution, the solution will have less round-off error than with Gauss-Jordan elimination.

HOW TO COMPUTE THE AREA UNDER A CURVE

Try to estimate the shaded area:

One approach is to divide it into trapezoids:

Denote the total area within the 4 trapezoids as S_4. If you use 8 thin trapezoids (instead of 4 fat ones), you'll get a more accurate estimate, called S_8. And S_{16} will be still more accurate. Each time you double the number of trapezoids, you'll increase the accuracy of the estimate— until you start getting into round-off problems, caused by taking the sum of many small numbers. Suppose the most accurate estimate you can get before round-off swamps you is S_{32}.

According to mathematical theory, if you multiply the number of trapezoids by n, you tend to divide the error by n^2 (until round-off sets in). In other words,

$$S_{nk} - \text{True area} \cong \frac{S_k - \text{True area}}{n^2}$$

(The symbol "≅" means "approximately equals".) For example, if you let n be 2, you have:

$$S_{2k} - \text{True area} \cong \frac{S_k - \text{True area}}{4}$$

Solving that equation for True area, you get:

$$\text{True area} \cong S_{2k} + \frac{S_{2k}-S_k}{3}$$

For example, you can estimate the True area to be $S_{32} + \frac{S_{32}-S_{16}}{3}$. That estimate will be more accurate than S_{32}, while still avoiding the round-off error of S_{64}. That estimate is called T_{32}. In general, $T_{2k} = S_{2k} + \frac{S_{2k}-S_k}{3}$, and is a better estimate than S_{2k}.

The estimate is not the exact area, although it's a close approximation. If you multiply the number of trapezoids by n, you tend to divide the error of T by n^4. In other words:

$$T_{nk} - \text{True area} \cong \frac{T_k - \text{True area}}{n^4}$$

For example, if you let n be 2, you get:

$$T_{2k} - \text{True area} \cong \frac{T_k - \text{True area}}{16}$$

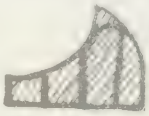
Solving that equation for True area, you get:

$$\text{True area} \cong T_{2k} + \frac{T_{2k}-T_k}{15}$$

Therefore, in our example, you can estimate the True area to be $T_{32} + \frac{T_{32} - T_{16}}{15}$, where T_{32} is $S_{32} + \frac{S_{32}-S_{16}}{3}$, and T_{16} is $S_{16} + \frac{S_{16}-S_8}{3}$. That estimate is called U_{32}.

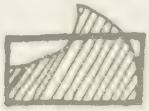
The U estimate is not the exact area either, although it's a close approximation. If you multiply the number of trapezoids by n, you tend to divide the error of U by n^6. Imitating the previous reasoning, you're led to a V estimate, which has order n^8, and leads to a W estimate, which has order n^{10}, and leads to an X estimate, etc. This use of successively better estimates is called *Romberg integration*.

A less intelligent approach is to divide the area into parabolas:



Notice that the two parabolas require the same number of evaluations as four trapezoids. If you use n parabolas (which require as many evaluations as 2n trapezoids), you'll get a more accurate estimate than S_{2n}. In fact, you'll get T_{2n}. So parabolas give the same answer as using Romberg's method halfheartedly (computing T but not U or V or W). The use of parabolas is called *Simpson's rule*.

A totally different approach to estimating the shaded area is *Gaussian quadrature*. The area is estimated to be the area of a rectangle that has the same width and a certain height:



To compute the height of the rectangle, you find the height of the shaded area at certain chosen points, and take a weighted average. The chosen points are irregularly spaced, unlike the points used in the trapezoid and Simpson rules; the spacing and weighting are concocted so that, by examining the height at just n points, you'll get the *exact* area if the curve is a "polynomial of degree ≤ 2n-1".

CHAPTER 10

CHANGING LIVES

YOUR CAREER

HOW TO BECOME AN EXPERT

To become a computer expert, you need three things: a computer to practice on, literature to read, and people to chat with. Let's look at the details. . . .

A computer to practice on. If you can afford it, get an IBM PC. If you're poor, get a VIC-20, which costs just $85. If you can't afford even a VIC-20, find 8 friends, and get each of them to chip in $10: altogether, your group will have $90, which is enough to buy a VIC-20! Divide the 9 of you into 3 trios, and rotate the VIC-20 from trio to trio every day: that way, you'll get to use the VIC-20 every third day.

Literature to read. Begin by reading both volumes of *The Secret Guide to Computers*. Then read the manuals that came with your computer. Then find out what's new in the computer industry, by subscribing to the four best computer magazines, which are *Infoworld*, *Creative Computing*, *Popular Computing*, and *Byte*.

A one-year subscription costs $31 for Infoworld, $25 for Creative Computing, $15 for Popular Computing, and $21 for Byte. So altogether, you'll spend $92 for all four.

Infoworld is at 617-879-0700 or 800-343-6474. Creative Computing is at 201-540-0445 or 800-631-8112. Popular Computing and Byte are at 603-924-9281.

To find other good magazines, newspapers, and books about computers, browse in your local computer store and bookstore.

The best source of computer books in New York City is the McGraw-Hill Bookstore (1221 Avenue of the Americas, 212-997-1221); the best source of computer books in Los Angeles is the Opamp Bookstore (1033 N. Sycamore Ave., 213-464-4322). In most other parts of the country, the best source of computer books is your local branch of B. Dalton Bookseller.

If you live near Boston, the best source of computer magazines is the kiosks in the middle of Harvard Square.

People to chat with. Whenever you have a question about computers, you can call me on the phone. Another way to communicate with computerists is to join a computer club. If you live in Connecticut, join the Connecticut Computer Society by phoning 203-233-8594 or 203-233-3373; it costs $20/year. If you live near Boston, join the Boston Computer Society by phoning 617-367-8080; it costs $24/year. If you live elsewhere, ask your local computer store or high-school computer department about computer clubs; if there aren't any, start one yourself!

Another way to find people to chat with is to take a computer course and chat with the teacher and your classmates.

For inexpensive computer courses that are worthwhile, take your high school's "adult education" evening courses, and take the courses offered by your local community college. Though such courses aren't fancy, at least they're inexpensive.

I used to be a computer missionary: I drove around New England and the Middle Atlantic States, in a van containing 30 computers that I owned, and gave courses at little or no charge. Many heads of the computer industry got their training from me at those courses. Unfortunately, because I must answer so many phone calls now, and because The Secret Guide to Computers (which needs continuous updating) has grown so large, I don't have much time left for giving courses anymore. I hope to give a few such courses again someday, and to offer the courses for free; to get on the mailing list, use the coupon on the back page.

The fanciest courses I gave were at Wesleyan University and were called the Summer Computer Institute. Although I don't have enough time to run such an Institute anymore, several members of my faculty are planning to run imitations that will be almost as good as the original—and maybe they'll even be better!

I'm particularly impressed by the initiation that will be run by Cindy Herriman, a top-notch teacher who's extensively researched how to further improve the Summer Computer Institute. She's managed to further lower the cost, to make her version of the Summer Computer Institute the most cost-effective ever. To find out about her wonderful program, which she's holding at the University of New Hampshire, phone her at 603-436-1608, day or evening. Although I'm not involved in the teaching or administration or financing of her program, I applaud her efforts to continue and further enhance the tradition of highly intensive courses at rock-bottom prices.

Another excellent program will be run by Larry Tovena at Wesleyan University in Connecticut, though at a somewhat higher price. For details, phone him at 617-482-5432.

So far, Cindy's program seems to be a bit better than Larry's. But as summer approaches, and more details about Cindy's and Larry's programs materialize, I can give you more definitive advice: phone me at 617-266-8128.

If you hear of any other top-notch programs at low prices, tell me, so I can tell others.

HOW TO GET A JOB

To become a lawyer, you must graduate from law school and pass the Bar Exam. But becoming a computer expert is different: there's no particular program you must graduate from, and there's no particular exam you must pass. There's no particular piece of paper that "proves" you're an expert or even competent.

You can get a job in the computer industry even if you've never had any training. Your job will be sweeping the floor.

If you want to become one of the top experts in the field, you must study, study, study, day and night. You must read many manuals, textbooks, guidebooks, magazines, newspapers, and newsletters about computers. You must get hands-on experience, using many kinds of computers and their many operating systems and languages. You should personally use many word-processing programs, many electronic spreadsheets, and many data-management programs, and many educational programs for kids. You should use many kinds of printers, many kinds of disk drives, and many kinds of telephone hookups. You should also study the human problems of dealing with computers. No matter how much you already know, you should always be learning more, more, more.

I did a survey of computer experts. The survey found that the average computer expert spends two hours per day reading more about computers. Yes, the average computer expert spends two hours per day, just to patch holes in the expert's background, and to find out what new happened in the computer industry that day! In addition to those two hours, the expert spends many more hours practicing what was read, and sharing new information verbally with others.

As a computer expert, you can choose your own hours, but they must be numerous: if your interest in computers lasts just from 9 AM to 5 PM, you'll never become a computer expert.

To break into the computer field, you can use five methods. . . .

1. The most traditional method is to go to college and get a Ph.D. or M.A. in computer science. Unfortunately, that takes a lot of time.

2. A quicker route to success is to learn enough about microcomputers so that you can get a job in a store that sells them. As a salesperson, you'll be helping people decide which computer to buy; you'll be acting as a consultant. The store will probably give you permission to take hardware and software and literature home with you, so that you can study and practice new computer techniques every evening and become brilliant. If you wish, you can even do some moonlighting, by helping your customers program their computers. After working at the store for several months, you'll have enough knowledge and experience and a good enough reputation and enough contacts with customers so that you can set yourself up as an independent consultant. You can call up your former customers and become their advisor and programmer. Or, if you prefer, set up your own store.

3. A third way to break into the field is to take a non-computer job but gradually turn it into a job that has computer responsibilities. For example, if you're a math teacher, try to convince the principal to let you try teaching a computer course or help run the school's computer club. If you're a typist, try to convince your boss to let you use a word processor. If you're a clerk, try to convince your boss to let you use a data-management system and an electronic spreadsheet to help manage your work more quickly. Keep your current job but expand it to include new skills and opportunities, so that you gradually become a computer expert.

4. The most entrepreneurial way to break into the field is to keep your current job but spend your evenings at home writing computer programs, which you then try to sell to software publishers and to friends.

5. The remaining way to break into the field is to get a job in a computer company, as a janitor or clerk, and gradually move up, by using the company's policy of giving a little free training to employees.

Many companies phone me when they're looking for computer experts. If you think you're an expert and can demonstrate your expertise, I'll be glad tc pass your name to employers.

Occasionally, I even have job openings here at The Secret Guide to Computers. Feel free to ask. Although some of the jobs here are mundane, a nice fringe benefit is that you get to play with my 30 computers and oodles of software packages, and even take them all home with you. You can also choose your own hours: work whenever you please! After you work here a few months and do your job well, I'll be glad to give you an excellent reference that will help you get an even nicer job elsewhere.

HOW MUCH TO CHARGE

If somebody's interested in hiring you to be a programmer or consultant, you must decide how much to charge. If this is your *first* such job, I recommend that you charge very little, because your main purpose on your first job should not be money; instead, your main purpose should be to gain experience, enhance your reputation, and find somebody you can use as a reference and who can give you a good recommendation. You want to convince your first employer that you're the best bargain he ever got, so that he'll be wildly enthusiastic about you and give you a totally glowing recommendation when you go to seek your second job.

If you can't find anyone who's willing to pay you, work for free, just so that you can put on your resume that you "helped computerize a company". After such an experience, you should have no trouble finding a second job that pays better.

Although your first computer job might pay little or nothing, it gets your foot in the computer industry's door. After your first job, your salary will rise rapidly, because the most valuable attribute you can have in this field is *experience*.

Experienced experts are in short supply and are therefore paid astronomical salaries. On the other hand, there's a *surplus* of "kids fresh out of college" who know nothing. So consider your first job to be an extremely valuable way to gain experience, even if the initial salary is low. When applying for your first job, remember that you're still unproven, and you should be thankful that your first employer is willing to take a risk on you.

After several months on the job, when you've thoroughly proved that you're worth much more than you're being paid, and your employer is thoroughly thrilled with your performance, gently ask your employer for a slight raise. If he declines, continue working at that job, but also keep your eye open for a better alternative.

A fundamental rule of contract negotiation is: never make a large commitment. For example, suppose somebody offers to pay you $10,000 if you write a fancy program. Don't accept the offer; the commitment is too large. Instead, request $1,000 for writing a stripped-down version of the program. After writing the stripped-down version, wait and see whether you get the $1,000; if you get it without any hassles, then agree to make the version slightly fancier, for a few thousand dollars more. That way, if you have an argument with your employer (which is common), you've lost only $1,000 of effort instead of $10,000.

Arguments between programmers and employers are common, for six reasons. . . .

1. As a programmer, you'll probably make the mistake of underestimating the time for debugging the program, because you'll tend to be too optimistic about your own abilities.

2. Your employer won't be precise enough when he tells you what kind of program to write. You'll write a program that you *think* satisfies the employer's request and then discover that the employer really wanted something slightly different.

3. Your employer will forget to tell you about the various "strange cases" that the company must handle, and which will require extra "IF" statements in your program.

4. When your employer finally sees your program working, he'll suddenly think of extra things he'd like the program to do, and which will require extra programming effort from you.

5. When the program finally does everything that the employer expects, he'll want you to train his staff in how to use the program and the computer. If his staff has never dealt with computers before, the training period could be quite lengthy. He'll also want you to write a manual about the program, and to put the manual into the company's library.

6. After the company begins using the program, the employer will want you to make additional changes, and might even expect you to make them at no charge.

To minimize those six kinds of conflicts, be honest and kind to your employer. Explain to him that you're worried about those six kinds of conflicts, and that you'd like to chat about them *now*, before either you or he makes any commitments. Then make a small commitment for a small amount of payment for a small amount of time, and make sure that both you and the employer are happy with the way that small commitment worked out, before attempting any larger commitments.

EDUCATION

CURRICULUM

When should kids start learning about computers?

Programs have been developed even for kids in nursery school! For example, you can get "alphabet fun" programs: when the kid presses the A key, pictures of apples appear all over the screen; when the kid presses B, the screen is filled with bears; C gives cats, etc. To make the program fun, the pictures on the screen are *animated*; they dance

Kids should start writing simple programs in BASIC when they're in the third grade. (The brightest kids can start even younger!) Before the third grade, the typical kid should learn how to run other people's programs (by typing the word RUN) and maybe should learn LOGO, which is a language for beginners that's easier than BASIC.

Which kids should take computer courses?

All kids should be exposed to the computer. All kids should have the opportunity to press the buttons, type the word RUN, and do other fun things. All kids should deal with the computer *before* entering high school. The introductory instruction should be broad: it should dip into BASIC programming, hardware jargon, application areas (such as word processing), and social effects. The introduction is important for *all* kids, regardless of mathematical ability; remember that most computer programming requires hardly any math. Kids who are "slow" or who "hate school" should be included, since the computer often acts as a catalyst for making such students "turn on" to school and "tune in"; LOGO has been particularly effective at that. If your school has too few computers to permit an extensive program, the best compromise is to wheel the computers from classroom to classroom, so that each kid gets to spend at least a few minutes with the computer each semester. Kids who want to go beyond introductory concepts should be allowed to join an after-school computer club.

Which language should kids learn to program in?

More programs have been written in BASIC than in any other computer language. Therefore, a person who doesn't understand BASIC is "out of touch" with reality and is a computer illiterate. Every kid should learn BASIC before graduating from high school The youngest kids might also want to experiment with LOGO (which lets you draw pictures more easily than BASIC); the oldest kids might also want to experiment with PASCAL (which is more "sophisticated" than BASIC). But BASIC is the most "practical" language to learn, because it can handle a wider variety of applications than LOGO and PASCAL. (LOGO can't handle random-access files well; PASCAL can't handle output formats well.) Also, BASIC is less expensive than LOGO and PASCAL. (LOGO requires more memory; PASCAL requires more memory and also requires disk drives.)

What should a computer course emphasize?

The course should emphasize hands-on programming, with fun applications to a wide variety of topics. The course should *not* be restricted to math and science; in fact, less than half of the programming examples should involve math or science; the majority of the examples should involve the arts, business, word processing, etc. If the computer course is taught by a math teacher, the school's principal should check that the teacher doesn't spend too much time talking about math.

In the "computer curriculum", how important are music and graphics?

Any computer for kids should play music and draw graphics (preferably in color), because music and graphics create fun and maintain the kids' interest. Any lectures on computer programming should include a discussion of how to program music and graphics: besides being fun, such a discussion emphasizes that computers are not "just for numbers", and also illustrates dramatically and clearly the effects of programming concepts such as FOR...NEXT loops.

In a computer course, what homework should be assigned?

The homework should be to write a computer program. To make that practical, the school must have enough computers to handle all the kids. Although the teacher should assign some standard exercises, the kids should also be encouraged to invent their *own* programming projects.

In what order should computer topics be taught?

The course should begin with hands-on experience: the kids should write elementary programs (in BASIC or LOGO) and run programs that others wrote. As the course progresses, the programming examples tend to become more abstract and sophisticated; to give the kids a "breather" from such heavy stuff, frequently insert light-hearted topics such as "video games", "computerized art", "effects of computers on society", "word processing", "kinds of hardware", "computer companies", and "computer jobs".

EDUCATIONAL APPLICATIONS

How can the computer help teach English?

While trying to write a program, the kid learns the importance of punctuation: the kid learns to distinguish between colons, semicolons, commas, periods, parentheses, and brackets. The kid also learns the importance of spelling: if the kid misspells the word PRINT or INPUT, the computer gripes. The kid learns to handle long words while wading through computer manuals.

Some kids "hate to write English compositions". The computer can change that attitude! If you let a kid use an easy word-processing program (such as Superscript!) the kid suddenly discovers that writing an English composition can be fun! The composition suddenly becomes "electronic"; it appears on the television screen! And revising the composition can be even *more* fun, since the kid gets to use all the nifty "editing" keys on the keyboard. The whole experience becomes as much fun as a video game. A high-quality word processor also corrects the kid's spelling without forcing the kid to endlessly thumb through the dictionary, it even corrects the kid's grammar and style. Watching the computer correct the spelling, grammar, and style is educational and fun.

To make the kid understand why parts of speech (such as "nouns", "verbs", and "adjectives") are important, give the kid a computer program that writes sentences by choosing random nouns, random verbs, and random adjectives. Then tell the kid to invent his *own* nouns, verbs, and adjectives, feed them into the program, and watch what kind of sentences the program produces now.

Young kids enjoy a program called *Story Machine*. It gives you a list of nouns, verbs, adjectives, and other parts of speech that you can use to build a story. You type the story, using any words on the list. As you type the story, the computer *automatically illustrates it!* For example, if you type, "The boy eats the apple," your screen will automatically show a picture of a boy eating an apple! If you type *several* sentences, to form a longer story, the computer will automatically illustrate the entire story and produce an animated cartoon of it! The program also criticizes your story's structure. For example, if you say "The boy eats the apple" but the boy isn't near the apple yet, the program will recommend that you insert a sentence such as "The boy runs to the apple" beforehand. The program comes on a $25 disk from Spinnaker Software (215 First St., Cambridge, MA 02142, phone 617-868-4700). To run the program, you need an Apple or an IBM Personal Computer.

How can the computer help teach music?

Atari sells a program that lets the student compose music easily. The program begins by showing a colorful picture of the musical staff: it shows the treble clef and the bass clef. If the student types "C5Q", the computer displays a C in octave 5 that's a quarter note. It also plays the note (by using the Atari's built-in music synthesizer). If the student then types "FS4H", the computer displays an F sharp in octave 4 that's a half note; the screen shows the entire melody (it shows both the C and the F sharp); and the computer plays the entire melody (it plays the C again, followed by the F sharp). Every time the student adds a note, the computer plays the entire melody from beginning to end— until the student has completed a "measure". The computer combines several measures to form a phrase, combines phrases to form a melodic voice, and combines voices to form harmony. It's lots of fun! If the student makes a mistake, the student uses the editor to eliminate it. Radio Shack offers a similar program for the Radio Shack Color Computer; the Radio Shack version is even easier to use than Atari's, but lacks harmony.

Those programs by Atari and Radio Shack come as ROM cartridges; just pop them into your Atari or Radio Shack computer. They turn your computer into a "music processor" instead of a "word processor".

How can the computer help teach history?

The computer can make history become "alive", by throwing the student into the middle of a historical situation. For example, a graduate of my intensive teacher-training institute wrote a program that says, "It's 1910. You're Kaiser Wilhelm. What are you going to do?" Then it gives you several choices. For example, it asks "Would you like to make a treaty with Russia?" If you answer "yes", the computer replies, "Russia break the treaty. *Now* what are you going to do?" No matter how you asnwer the questions, there are only two ways the program can end: either "You've plunged Europe into a World War" or "You've turned Germany into a second-rate country". After running that program several times, you get a real feeling for the terrible jam that the Kaiser was in, and you begin to pity him. Running the program is more dramatic than reading a book about the Kaiser's problems, because the program forces you to step into the Kaiser's shoes and react to the Kaiser's surroundings: you are there. When you finish running the program, you feel you've lived another life, the life of a 1910 Kaiser.

Such a program is called a historical *simulation*, since it makes the computer *simulate* (imitate) a historical event.

How can the computer help teach about current events?

The best way to teach about current events is through simulation.

For example, when California's Governor Brown had trouble controlling medflies, teachers wrote programs that began by saying, "You're Governor Brown. What are you going to do?" (One of the programs was even called "Medfly Mania".)

The best way to encourage the student to analyze the conflict between Israel and the Arabs is to tell the student to run a program that begins by saying "You're Israel's Prime Minister Begin" and then run a program that begins by saying "You're the PLO's leader Yassir Arafat"; by running both programs, the student learns to take both sides of the argument and understand the emotions of both leaders. Such a program could help warring nations understand each other and thereby bring peace!

When Three-Mile Island almost exploded, teachers wrote a program that says "You're in the control room at Three-Mile Island". Your computer's screen shows a high-resolutio color picture of the control room. Your goal is to make as much money as possible for the electric company, without blowing the place up. You can buy two versions of the progra one's called just "Three-Mile Island"; the other's called "Scram". To teach kids about Three-Mile Island, it's easier to buy the program than to get permission from parents to "take the kids on a field trip to Three-Mile Island" (which also requires that you sit on a bus while listening to 99 choruses of "bottles of beer on the wall" and worrying about kids who get lost at Three-Mile Island).

The best way to teach economics and politics is to give the student a program that sa "You're running the country" and then asks the student to input an economic and political strategy. At the end of the program, the computer tells how many years the student lasted in office, how well the country fared, and how many people want to assassinate him.

The best way to learn anything is "by experience". Computer simulations let the student learn by "simulated experience", which condenses into a few minutes what would otherwise require many *years* of "natural experience".

How can the computer help teach biology?

The computer can do genetics calculations: it can compute the probabilities of having various kinds of offspring and predict how the characteristics of the population will shift over time.

The computer can handle taxonomy: it can classify different kinds of animals and plants. The computer asks you a series of questions about an organism and finally tells you the organism's name. One of the most popular programs is a game called "Animals", which lets the student teach the computer which questions to ask.

To teach ecology, a graduate of my teacher-training institute wrote a simulation program that begins by saying, "You're the game warden of New Jersey. What are you going to do?" For example, it asks how many weeks you want the deer-hunting season to last. If you make the hunting season too long, the hunters kill all the deer, and the deer-loving environmentalists hate you. On the other hand, if you make the deer-hunting season too *short*, the *hunters* hate you; moreover, the deer overpopulate, can't find enough to eat, and then die of starvation, whereupon *everybody* hates you. Your goal is to stay in office as long as possible.

When Dartmouth College (which for centuries had been all-male and rowdy) suddenly became co-ed in 1971, its biology department realized the importance of teaching about birth control. The professors wrote a program that asks how old you are and which birth control method you wish to use this year. You have 9 choices, such as pill, diaphragm, IUD, condom, rhythm method, and "Providence". After you type your choice, the computer computes the probabilities and may print (if you're unlucky) ***BOY*** or ***GIRL***. The computation is based, as in nature, on a combination of science and chance (random numbers). Then it asks your strategy for the next year. The program continues until it finally prints ***MENOPAUSE***. Using the program, you can explore how different strategies will result in different numbers of children. It's safer to experiment with the program than to experiment on your body. It's also faster. (But maybe it's not as much fun?)

How can the computer help teach chemistry?

You can get programs that balance chemical equations. You can also get programs that classify organic compounds.

How can programs that tutor, drill, and test students be made exciting?

The programs should use the same techniques that make video games exciting. Specifically, the programs should include graphics (preferably in color) and animation, should require the student to answer quickly, and should display a running total of the student's points, so that each time the student answers a question correctly the score on the screen increases immediately. At the end of the educational game, the computer shouldn't say "excellent" or "fair" or "poor"; instead, the computer should simply state the total number of points accumulated (which should be in the thousands) and then ask whether the student would like to try again, to increase the score. If the student's score is exceptionally high, the computer should reward the student by giving lots of praise and by storing the student's name on the disk; if the student's score is low, no criticism should be given other than asking "Would you like to try again?"

How many computers should the school buy?

The more the better! Whatever you can afford!

Which computer is the best for teaching programming?

The IBM Personal Computer is the best for teaching programming, because it has the most sophisticated version of BASIC and also has the pleasantest and easiest program editor. But since the IBM's price is prohibitively high, I recommend that you buy a cheaper computer instead. The cheapest computer that's good for beginners is the Commodore VIC, which costs just $85 from discount dealers; so for just $85 plus the price of a discount black-and-white TV ($70), you can have a complete computer system. Instead of spending $3000 for one IBM or a souped-up Apple, you should spend the $3000 for 19 VIC's and 19 TV's; that way, you can make 19 kids happy instead of just 1. For just $3875, you can buy enough VIC's to give a personal computer to 25 students in your class!

Another alternative is to buy the Radio Shack Color Computer with 16K and Extended BASIC. It's easier to program than the VIC (and also handles more advanced commands) but unfortunately costs twice as much.

For students who outgrow the VIC, you'll need the Radio Shack Color Computer (or even a Radio Shack model 4 or Apple or IBM Personal Computer).

Should the school buy a variety of different computers, or stick to a single brand?

The students should see a wide variety of computers, because each computer has its own strengths and weaknesses. For example, students should see the pleasantest file-management program, which is called "PFS" and runs only on Apple, Radio Shack, and IBM computers; students should see the pleasantest word-processing program, which is called "Super Scripsit" and runs well only on the Radio Shack model 4 or 4P; students should see the most dramatic outer-space game, which is called "Space Invaders" and runs only on the Atari; students should see the best built-in editor for BASIC, which runs only on the IBM Personal Computer.

The school should decide on a "main" brand of computer (to simplify the lectures about programming) but also buy samples of the other brands (for demonstrations and for advanced students).

How can teachers get software without paying for it?

If you're a teacher, tell your hot-shot students to write the software for you. Your students will love the opportunity to work on a project that's useful. Tell the students that if their software is good you'll help them figure out how to sell it.

Should kids be allowed to play video games on the school's computers?

Every kid should have the experience of playing video games briefly, because they're fun, encourage speed and agility, reward self improvement, create a positive attitude towards computers and technology, lead the kid to thinking about strategies and programming methods, and provide examples of the best programs ever invented. On the other hand, kids should be discouraged from spending excessive time on games. Game-players should be given lower priority than other kids who want to use the computers. To do that, you can restrict game-playing to just a few of the computers, or restrict game-playing to just a few times of the day, or require game-players to leave whenever non-game-players want to use their computers. By charging a small fee for game-playing, you can collect enough money to buy more computers.

Which room should contain the computers?

The safest place to put the computers is in the library. That reduces the chance of theft, encourages disks to be checked out like books, and assures that the computer lab is run by a humanities-oriented librarian instead of a narrow-minded mathematician. Most librarians know how to run audio-visual equipment and how to communicate with large databases and therefore *don't* fear technology. Since librarians enjoy the humanities (especially reading) and nevertheless are comfortable with scientific technology, librarians are the ideal choice for running a computer center that meets the need of the *whole* school. And the library is the only place in the school where all the students and faculty can feel comfortable— except for the cafeteria.

Try moving a few low-cost computers into the cafeteria, for students to use during lunch and during study breaks. That will increase the computers' visibility and also turn lunch into an intellectual affair. With adequate supervision, you can overcome the cafeteria's dangers (theft, food fights, and spilled drinks).

How can the computers be supervised inexpensively?

You can get parents to volunteer. Many parents would love the opportunity to work in a computer environment, in the hope of entering a full-blown computer career later.

If the students in your school's computer club act responsibly, you can turn the club into a "computer-service organization" that helps teach the rest of the school about computers. Members of the computer-service organization can mention such service on their resumés, which will help them get into college.

Give a speech to all students: tell them that when they're using the computers they should help each other. Encourage teamwork.

Should students be allowed to use the administration's computer?

The administration's computer handles the school's budget, payroll, schedules, attendance records, and report cards. Students shouldn't have access to the disks containing that information. But students *should* be allowed to occasionally use the administration's computer (provided the students use special "student disks" instead), because the students should have the opportunity to see how a *big* computer operates. The brightest and most trustworthy students might even help the administration write some programs (though the students shouldn't be given access to the real data).

ANCIENT HISTORY

BEFORE COMPUTERS

The first programmable computers were invented in the 1940's. Before then, there wasn't much to talk about. People had invented the abacus, then the adding machine, and then the slide rule, but those gadgets were far simpler than computers.

The history of punched cards is weird. The cards were first used for *weaving tapestries*. Where the cards had holes, rods could move through the cards; those moving rods in turn made other rods move, which caused the threads to weave pictures. That machine was called the *Jacquard loom*.

CHARLES BABBAGE

Charles Babbage was a wild-eyed English mathematician who, in the 1800's, believed he could build a fancy computing machine. He convinced the British government to give him lots of money. Then he bilked the government for more. Many years later— and many British-pounds later— he still hadn't finished his machine. So he dropped the idea and— can you believe this?— tried to build an even *fancier* machine. Needless to say, he didn't finish that one either. So you might say his life was a failure— and an expensive one, for the British government.

But Charlie (as I shall call him) is admired by all us computerniks (in spite of his face, which was even sterner than Beethoven's), because he was the first person to realize that a computing machine must be composed of an input device (he used a card reader), a memory (which he called "The Store"), a central processing unit (which he called "The Mill"), and an output device (he used a printer).

LADY LOVELACE

Women's Libbers will kill me if I don't mention Charlie's side-kick, Lady Lovelace. (No, she's not related to Linda.) She was one of Charlie's great admirers, but he never noticed her until she translated his stuff. And boy, it was impossible for him *not* to notice her translation. Her "footnotes" to the translation were three times as long as what she was translating!

She got very intense. She wrote to Charlie, "I am working very hard for you; like the Devil in fact; (which perhaps I am)."

The two became love-birds, although he was old enough to be her father. (By the way, her father was Lord Byron, the poet. She was Lord Byron's only "official" daughter. He had several other daughters, but they were illegitimate.) Some people argue that she was actually brighter than Charlie, despite Charlie's fame. She was better at explaining Charlie's machines and their implications than Charlie was. Some people have dubbed her, "the world's first programmer".

She stunned all the men she met. She was so bright and... a woman! Here's how the editor of *The Examiner* described her (note the pre-Women's-Lib language!):

"She was thoroughly original, and her genius, for genius she possessed, was not poetic, but metaphysical and mathematical. With an understanding thoroughly masculine in solidity, grasp, and firmness, Lady Lovelace had all the delicacies of the most refined female character. Her manners, her tastes, her accomplishments, were feminine in the nicest sense of the word; and the superficial observer would never have divined the strength and the knowledge that lay hidden under the womanly graces. Proportionate to her distaste for the frivolous and commonplace was her enjoyment of true intellectual society. Eagerly she sought the acquaintance of all who were distinguished in science, art, and literature."

Eventually, she went mad. To prevent her from banging her head, they lined her room with mattresses. Nevertheless, she died gruesomely, at the ripe young age of 36, the same age her father died. (I guess premature death was popular in her Devilish family.)

I wish the Women's Lib press would pick a different idol than Lady Lovelace. She was *not* the most important woman in the history of computing. Far more important were Grace Hopper (who, in the 1950's, invented the first programming languages) and Jean Sammet (who headed the main committee that invented COBOL, and who is generally considered to be *the* world's expert on the history of programming languages, and who has been president of the U.S. organization of computer professionals, which is called the ACM). Lady Lovelace was second-string to Babbage. Grace Hopper and Jean Sammet are second-string to *nobody*. But unfortunately, Hopper and Sammet lead less racy lives, so journalists ignore them. Also, since Hopper is an Admiral in the Navy (bet you didn't know the Navy has lady Admirals!), she doesn't enchant some of us doves. Nevertheless, every time she steps in front of an audience, she gets a standing ovation, because all of us realize how crucial she was to what computers are today.

But I'm straying from my story....

HERMAN HOLLERITH

The U.S. Bureau of the Census takes its census every ten years. For the 1880 census, the guys at the Bureau had a hard time: it took them *7 years* to tabulate the results of the census. In other words, they didn't finish until 1887. When they contemplated the upcoming 1890 census, they got scared: at the rate America was growing, they figured that tallying the 1890 census would take 12 years. In other words, the results of the 1890 census wouldn't be ready until 1902. So they held a contest, to see whether anyone could invent a faster way to tabulate the data.

The winner was Herman Hollerith. He was the first person to successfully use punched cards to process data.

Hermie (as I shall call him) was modest. When people asked him how he got the idea of using punched cards, he had two answers. One was, "Trains": he had watched a train's conductor punch the tickets. His other, more interesting answer was, "Chicken salad". After saying "Chicken salad", he would pause, and wait for you to ask the obvious question, "*Why* chicken salad?" Then he would tell his tale....

One day, a girl saw him gulping down chicken salad. She said, "Oh, you like chicken salad? Come to my house. My mother makes excellent chicken salad." So he did. And her father was a head of the Census. (And he married the girl.)

By the way, Herman Hollerith hated one thing: spelling. In elementary school, he jumped out a second-story window, to avoid a spelling test.

In some versions of FORTRAN, every string must be preceded by the letter H. For example, instead of saying—

```
'DOG'
```

you must say:

```
3HDOG
```

The H is to honor Herman Hollerith.

Anyway, getting back to the story, in 1890 the Census used Hollerith's system, and used it again in 1900. But in 1910, the Census switched to a more advanced system, created by a Census Bureau employee named James Powers, who eventually left the Census to start his own company, which eventually merged into Remington Rand Sperry-Univac. Meanwhile, Herman Hollerith's own company merged into IBM. And that's how the first two computer companies got started in data processing.

WORLD WAR II

The first programmable computers were invented in the 1940's, because of World War II. They *could* have been invented sooner— most of the know-how was available several decades earlier— but you can't invent a computer unless you have big bucks for research. And the only organization that had big enough bucks was the Defense Department (which in those days was more honestly called the "War Department"). And the only event that was big enough to make the War Department spend that kind of money was World War II.

Of course, the Germans did the same thing. A German fellow named Konrad Zuse built computers which were in some ways better than the American ones. But since the Germans lost the war, you don't hear much about old Konrad anymore. Fortunately, throughout World War II the German military ignored what he was doing.

Most of the computers in the 1940's were invented at universities, usually with funds from the War-Defense Department. Some of the most famous computers were the Mark I (at Harvard, with help from IBM), the ENIAC and the EDVAC (both at the University of Pennsylvania), the Whirlwind (at M.I.T.— the Massachusetts Institute of Technology) and the Ferranti Mark I (at the University of Manchester, in England). Which of those computers deserves to be called "the first programmable computer"? The answer's up for grabs. Each of those machines had its own peculiar hang-ups, and required several years of debugging before it actually worked decently.

Each of those computers was, as they say in the art world, a "signed original". No two of those computers were alike.

THE FIRST GENERATION
1951-1958

It all started with UNIVAC

The first computer to be mass-produced was the UNIVAC I, in 1951. It was made by two guys, named Eckert and Mauchly— the same guys who built the ENIAC and EDVAC at the University of Pennsylvania. (Mauchly was an instructor there, and Eckert was the graduate student who did the dirty work.) While others at the school were helping build the EDVAC, Eckert and Mauchly left and formed their own company, which invented and started building the UNIVAC. But by the time the UNIVAC was completed, the Eckert-Mauchly company had merged into Remington Rand (which later merged into Sperry-Rand, which sprouted a division which today is called Sperry-Univac).

Because the UNIVAC I was so important, historians call it the beginning of the "first generation". As for computers before UNIVAC— historians disparagingly call them the "zeroth generation".

So the first generation began in 1951. It lasted through 1958. Altogether, from 1951 to 1958, 46 of those UNIVACs were sold.

46 might not sound like many. But remember: in those days, computers were very expensive, and could do very little. Another reason why only 46 were sold is that newer models came out, such as the UNIVAC 1103, the UNIVAC 80, and the UNIVAC 90. But the biggest reason why only 46 of the UNIVAC I were sold is IBM.

The rise of IBM

Although IBM didn't begin mass-marketing computers until 1953— two years after UNIVAC— the IBM guys were much better salesmen, and soon practically everybody was buying from IBM. During the first generation, the hottest seller was the IBM 650. Yep, IBM sold hundreds and hundreds of them.

Of course, there were many smaller manufacturers too. People summarized the whole computer industry in one phrase: "IBM and the Seven Dwarfs".

Who were the dwarfs? They kept changing. Companies rapidly entered the field and then rapidly left, when they realized IBM had the upper hand. By the end of the first generation, IBM was getting 70% of the sales.

Primitive input and output

During the first generation, there were no terminals. To program the UNIVAC I, you had to put the program onto magnetic tape (by using a non-computerized machine), then feed the tape into the computer, then wait for the computer to spit out another magnetic tape, which you then had to run through another machine to find out what the tape said.

One reason why the IBM 650 became more popular was that it could read cards instead of tapes. It really liked cards. In fact, the answers came out on cards. To transfer the answers from cards to paper, you had to run the cards through a separate non-computerized machine.

Memory

At the beginning of the first generation, there was no RAM, no ROM, and no core. Instead, the UNIVAC's main memory was banks of liquid mercury, in which the bits were stored as ultrasonic sound waves. It worked slowly and serially, so the access time ranged from 40 to 400 microseconds per bit.

UNIVAC's manufacturer and IBM started playing around with a different kind of memory, called the Williams tube, which was faster (10 to 50 microseconds); but since it was less reliable, it didn't sell well.

In 1953, several manufacturers started selling computers that were much cheaper, because they used super-slow memory: it was a drum that rotated at 3600 rpm, giving an average access time of 17000 microseconds (17 milliseconds). (Today, some computers still use drums, but for *auxiliary* memory, not for *main* memory.) The most popular first-generation computer, the IBM 650, was one of those cheap drum computers.

The idea of core memory began in 1950. The first working models were invented in 1953 at M.I.T. and RCA, which argued with each other about who owned the patent. The courts decided in favor of M.I.T., so both RCA *and IBM* came out with core memory computers. Core memory proved so popular that even today many minicomputers and maxicomputers still use core, though RAM is finally starting to overshadow it.

Languages

During the first generation, computer programming improved a lot. During the early '50s, all programs had to be written in machine language. In the middle 50's, assembly language became available. By 1958, the end of the first generation, three major high-level languages had become available: FORTRAN, ALGOL, and APT.

Fancy programs

Programmers tried to make computers play a decent game of chess. All the attempts failed. But at IBM, Arthur Samuel had some luck with checkers. He got his first checkers program working in 1952, and then continually improved it, to make it more and more sophisticated. In 1955, he rewrote it so that it learned from its own mistakes. In 1956, he demonstrated it on national television. He kept working on it. Though it hadn't reached championship level yet, it was starting to look impressive.

Computer music scored its first big success in 1956, on the University of Illinois' ILLIAC computer. Hiller & Isaacson made the ILLIAC compose its own music, in a style that sounded pre-Bach. In 1957, they made the program more flexible, so that it produced many styles of more modern music. The resulting mish-mash composition was dubbed "The ILLIAC Suite", and put onto a phonograph record.

In 1954, IBM wrote a program that translated simple sentences from Russian to English. Work on tackling harder sentences continued— with too much optimism.

THE SECOND GENERATION
1959-1963

Hardware

Throughout the first generation, each CPU was composed of vacuum tubes. Back in 1948, Bell Telephone had invented the transistor, and everybody realized that transistors would be better than vacuum tubes; but putting transistors into computers posed many practical problems, which weren't solved for many years. Finally, in 1959, computer companies suddenly started delivering transistorized computers; that year marked the beginning of the second generation; sales of vacuum-tube computers immediately stopped.

All second-generation computers used core memory.

IBM

The *first* company to make transistors for computers was Philco, but the most *popular* second-generation computer turned out to be the IBM 1401, because it was business-oriented and cheap. IBM announced it in 1959, and started shipping it to customers in 1960.

Its core memory required 11.5 microseconds per character. Each character consisted of 6 bits. The number of characters in the memory could range from 1.4K up to 16K. Most people rented the 1401 for about $8,000 per month, but you could spend anywhere from $4,000 to $12,000 per month, depending on how much memory you wanted, etc.

Altogether, IBM installed 14,000 of those machines. IBM also installed 1,000 of a faster version called the 1410 (which required only 4.5 microseconds per character, had 10K to 80K, and rented for $8,000 to $18,000 per month, typically $11,000).

Even today, many of those 1401 and 1410 machines are hanging around, and companies are still trying to convert programs from the 1401 to the IBM 370.

Altogether, IBM produced six kinds of computers....

small business computers: the 1401, 1410, 1440, and 1460
small scientific computers: the 1620

medium-sized business computers: the 7010
medium-sized scientific computers: the 7040 and 7044

large business computers: the 7070, 7074, and 7080
large scientific computers: the 7090 and 7094

CDC

Several employees left Remington-Rand-Sperry-Univac and formed their own company, called the Control Data Corporation (CDC). During the second generation, CDC produced popular scientific computers: the 1604, the 3600, and the 3800.

Software

During the second generation, software improved tremendously.

The 3 major programming languages that had been invented during the first generation (FORTRAN, ALGOL, and APT) were significantly improved.

6 new programming languages were invented: COBOL, RPG, LISP, SNOBOL, DYNAMO, and GPSS.

The 3 most popular sorting methods were invented: the Shuffle Sort, the Shell Sort, and Quicksort.

Programmers wrote advanced programs that answer questions about baseball, write poetry, tutor medical students, imitate three-person social interaction, control a mechanical hand, prove theorems in geometry, and solve indefinite integrals.

THE DAWN OF THE THIRD GENERATION
1964-1967

The big bang

The third generation began with a big bang, in 1964. Here's what happened in 1964, 1965, 1966, and 1967....

Families

The first modern computer families were shipped. They were the CDC 6600, the IBM 360, the PDP-6, the PDP-8, and the PDP-10.

Of those families, the CDC 6600 ran the most quickly. The IBM 360 was the most flexible, and was also the only one that used integrated circuits. The PDP-6 and PDP-10 were the best for timesharing. The PDP-8 was the cheapest.

Here are the exact dates. In 1964, CDC began shipping the CDC 6600. In 1964, IBM announced it was going to create a new family. The IBM 360 shipments began in 1966. In 1964, DEC began shipping the PDP-6 (a maxicomputer); in 1965, DEC began shipping the PDP-8 (a minicomputer); in 1967, DEC began shipping the PDP-10 (a maxicomputer that's a souped-up PDP-6).

New languages

IBM announced it was going to create a new computer language, that would combine all previous popular languages, including FORTRAN, COBOL, and ALGOL. That new language was PL/I. It was designed especially for IBM's new computer, the 360. In 1966, IBM began delivering PL/I to customers.

Programmers invented the first successful languages for *beginners* using *terminals*. Those languages were called BASIC, JOSS, and APL. Here are the exact dates....

Dartmouth College invented the first version of BASIC in 1964, and significantly improved it in 1966 and 1967.

In 1964, the RAND Corporation invented JOSS, for the JOHNNIAC computer. In 1965, an improved version, called JOSS II, was put onto the PDP-6. Today people use three popular variations of JOSS: a souped up version (called AID), a stripped down version (called FOCAL), and a business-oriented version (called MUMPS).

IBM completed the first version of APL in 1965, and put it on an IBM 7090. In 1966, IBM wrote a much better version of APL, and put it on an IBM 360. In 1967, IBM began shipping APL to customers.

Stanford University invented the most popular language for statistics: SPSS.

Artificial intelligence

Researchers (who called themselves "experts in artificial intelligence") taught the computer to chat in ordinary English:

Bertram Raphael made the computer learn from conversations.
Daniel Bobrow made it use algebra to solve "story problems".
The Systems Development Corporation made it know everything in an encyclopedia.
General Electric made it answer military questions.
Ross Quillian made it find underlying concepts.
Joe Weizenbaum made it act as a psychotherapist.

Also, Richard Greenblatt wrote the first decent chess program. It was good enough to play in championship tournaments against humans.

THE ERA OF BOREDOM
1968-1974

Proof of boredom

As you can see, the first, second, and third generations— up through 1967— were exciting, full of action. But then, from 1968 to 1974, *nothing newsworthy happened.* That was the era of boredom.

During the era of boredom, progress was made, but it was gradual and predictable. Nothing dramatic happened.

Of course, nobody actually came out and said, "Life is boring." People phrased it more genteelly. For example, this passage was written in September 1971, by Robert Fenichel and Joe Weizenbaum, for their introduction to *Scientific American*'s computer anthology:

"Partly because of the recent recession in the American economy, but more for reasons internal to the field, computer science has recently relaxed its pace. Work has not stopped, but that the current mood is one of consolidation can scarcely be doubted. Just a few years ago, computer science was moving so swiftly that even the professional journals were more archival than informative. This book could not then have been produced without great risk of misfocus. Today it is much easier to put the articles that constitute this book— even the most recent ones— into context."

Since the first generation had lasted eight years (1951-1958), and the second generation had lasted four years (1959-1963), so people were expecting the third generation to last at most four years (1964-1967) and that some kind of "fourth generation" would begin about 1968. But it never happened.

The only "major" announcement around then came in 1970, when IBM announced it would produce a new line of computers, called the IBM 370, which would make the IBM 360 obsolete. But to IBM's dismay, many computer centers decided to hang onto the old 360, instead of switching to the 370. The 370's advantage over the 360 was little, until several years later, when IBM started developing 370 software that wouldn't run on the 360.

Since the difference between the 370 and the 360 was disappointingly small, not even IBM claimed that the 370 marked a fourth generation. Computer historians, desperate for something positive to say about the 370, called it the beginning of the "late third generation", as opposed to the 360, which belonged to the "early third generation".

The cruel fact is, in the entire history of computers, there was only one year all computer manufacturers acted together to produce something new. That year was 1959, when all manufacturers switched from vacuum tubes to transistors. Since 1959, we haven't had any consistency. For example, although the third generation began with a "big bang" in 1964, each manufacturer was banging on a different drum. IBM was proclaiming how great the IBM 360 would be, because it would contain integrated circuits; but other manufacturers decided to ignore integrated circuits for several eayrs, and concentrated on improving other aspects of the computer instead. For many years after the beginning of the third generation, CDC and DEC continued to use discrete transistors (a sign of the second generation) instead of integrated circuits.

Why?

The era of boredom happened for three reasons....

The preceding years, 1964-1967, had been so successful that they were hard to improve on.

When the Vietnam War ended, the American economy had a recession, especially the computer industry, because it had depended on contracts from the Department of Defense. In 1969, the recession hit its worst point, and computer companies had to lay off many workers. In that year, General Electric gave up, and sold its computer division to Honeywell. In 1971, RCA gave up also, and sold its computer division to Remington-Rand-Sperry-Univac.

The world wasn't yet ready for "the era of personal computing", which began in 1975.

Quiet changes

During the era of boredom, these changes occurred— quietly....

In 1970, DEC began shipping the PDP-11. The PDP-8 and PDP-11 became the most popular minicomputers— far more popular than IBM's minicomputers. So in the field of minicomputers, IBM no longer had the upper hand.

BASIC became the most popular language for the PDP-8 and the PDP-11 and most other minicomputers (except IBM's, which emphasized RPG). In high schools and business schools, most of the introductory computer courses used BASIC, instead of FORTRAN or COBOL.

Many businesses and high schools bought their own minicomputers, instead of renting time on neighbors' maxicomputers. The typical high-school computer class used a PDP-8. The richest high schools bought PDP-11's.

In universities, the social sciences started using computers— and heavily— for doing statistical analysis.

All new computer families used 8-bit bytes, so the length of each word was a multiple of 8 (such as 8, 16, 32, or 64). Most older computer families, invented before the era of boredom, had used 6-bit bytes, so the length of each word had been a multiple of 6: for example, the PDP-8 had a word of 12 bits; the PDP-10 and UNIVAC 1100 and General Electric-Honeywell computers had a word of 36 bits; and the CDC 6600 had a word of 60 bits. The IBM 360 was the first computer to use 8-bit bytes instead of 6-bit; during the era of boredom, all manufacturers copied that feature from IBM.

CRT terminals (which use T.V.-like screens) got cheaper and cheaper, until they were finally as cheap as hard-copy terminals (which use paper). Most computer centers replaced hard-copy terminals by CRT terminals, since CRT terminals were quicker, quieter, and could do fancy editing.

Use of keypunch machines decreased, because many computer centers replaced cards by CRT terminals.

Interest in new computer languages died. Most computer managers decided to stick with the old favorites— FORTRAN and COBOL— instead of switching to more progressive languages, such as PL/I. Switching to PL/I would have required many programmers to be retrained, and many programs to be rewritten; managers couldn't afford that.

Programmers made two last-ditch attempts to improve ALGOL. The first attempt, called "ALGOL 68", failed to win popular appeal, because it was too complicated. The second attempt, called "PASCAL", gained more support, but slowly.

Maxicomputers were given "virtual core"— disks that pretend to be core, in case you're trying to run a program that's too large to fit into core.

Semiconductor memory got cheaper and cheaper, until it was finally cheaper than core. Most manufacturers replaced core by semiconductor memory.

In 1971, Intel began shipping the first microprocessor (complete CPU on a chip). It was called the 4004, and had a word of only 4 bits. In 1972, Intel began shipping an improved version, called the 8008, whose word had 8 bits. In 1973, Intel began shipping an even better version, called the 8080.

THE ERA OF PERSONAL COMPUTING
1975 and afterwards

Altair

In January 1975, the cover of *Popular Electronics Magazine* featured a "complete computer for only $395". That computer was manufactured by a company called "MITS", which called the computer the "Altair 8800", because Altair is the main star in Star Trek. Many people read the article and bought the computer.

By today's standards, the computer was *not* complete. It came as a kit you had to assemble yourself, by using a soldering iron and following the wiring diagrams. It included *no* software— not FORTRAN, not BASIC, not even assembly language or an editor or an operating system. The only way to program it was to use machine language, and toggle in the program by flipping the switches on the front panel. The only way to get the answers was to look at the blinking lights, which gave the answers in binary. There was no ROM, and only $\frac{1}{4}$ of a K of RAM. There was no disk, no tape, no printer, and no terminal— not even a keyboard. When you bought the kit and tried to put it together, you usually found that parts were missing, and the diagrams contained errors. In short, it was just an annoying toy, for tinkerers. Moreover, MITS quickly raised the price from $395 to $439.

Gradually, MITS fixed its errors, and began offering software, RAM, and peripherals— at extra cost, so that the typical customer wound up paying several thousand dollars extra to get the computer to do anything useful. But what's a few thousand dollars, when most competitors were charging *tens* of thousands of dollars?

MITS got lots of attention from the press, which thought it might lead to "a computer in every home".

Imitations

Several new companies arose a tried to steal MITS's business, by imitating MITS. They offered equipment that was similar to MITS's but better and lower-priced. The most famous of these competitors was IMSAI, which bragged that its CPU, memory, etc., were interchangeable with MITS's but were more "rugged" and "durable" and cost less. Other early competitors were Processor Technology and Polymorphic Systems.

Just like MITS, all those companies used the 8080 CPU and the S-100 bus and offered BASIC, though the versions of BASIC were not quite the same. The hobbyist could do comparative shopping, buy different parts of his computer from different companies, and get the whole combination to work.

A few manufacturers tried to be different: Southwest Technical Products used the 6800 CPU; MOS Technology's KIM used the 6502 CPU; the Digital Group used the 8080 CPU but with an unusual bus. Those manufacturers fared less well than the ones that imitated the Altair.

The biggies

In 1977, two larger companies got into the act: Commodore announced it was going to make a computer called the "PET"; and Radio Shack announced it was going to make a computer called the "TRS-80". Today, the most popular personal computers are made by Radio Shack, Commodore, Apple, and IBM.

SPELLING C

ERRORS

Beginners often misspell these words:

Wrong	Right	Comment
COBAL	COBOL	"COBOL" means "COmmon Business-Oriented Language". "COBAL" is a co-ed who likes sex.
computor	computer	"Computer" is a machine or person that computes. "Computor" is a snobbish computer.
Dartmouth U.	Dartmouth Coll.	"Dartmouth Coll." is where BASIC was invented. "Dartmouth U." exists only in Utopia.
Epsom	Epson	"Epson" makes printers. "Epsom" makes salt.
hexidecimal	hexadecimal	"Hexadecimal" means "six and ten", which is "sixteen". "Hexidecimal" is icky.
hobbiest	hobbyist	A "computer hobbyist" is a person who likes computers. A "computer hobbiest" is even more hobbier.
imput	input	"Input" is what the computer takes "in". "Imput" is said only by "im"beciles.
silicone	silicon	"Silicon" is what you put in an integrated circuit. "Silicone" is what you put in your breast.
softwear	software	"Software" is the opposite of "hardware". "Softwear" is a negligée.
TRS-232	RS-232	"RS-232" means "Recommended Standard #232". "TRS-232" means you think Tandy's Radio Shack is God.
TSR-80	TRS-80	"TRS-80" stands for "Tandy's Radio Shack". "TSR-80" is a jerk who mumbles the alphabet backwards.

CHOICES

For these words, choose your favorite spelling:

IBM prefers "disc"; other companies prefer "disk".
To mean "half a byte", humble programmers prefer "nibble"; snobs prefer "nybble".
The British prefer "programer"; Americans prefer "programmer".

IN THE HOME

DREAMERS

Back in 1970, many programmers dreamed of putting computers in homes. Let's examine those dreams again. Some of those dreams have already come true: many American homes already contain computers.

I'm going to quote four of those early dreamers:

James Martin & Adrian Norman, who together wrote The Computerized Society (560 pages, published in 1970 by Prentice-Hall)

John Kemeny, who invented BASIC and wrote a prophetic book, Man and the Computer (151 pages, published in 1972 by Scribner's)

G. Cuttle, who wrote a prize-winning essay about using computers in homes (written in 1969)

Although the quoted passages were all written around 1970, they're still valid: they could have been written today; our society's dreams haven't changed.

WORK AT HOME

Cuttle said:

It may be more economic for companies to subsidize home "communications rooms" for their employees than to rent expensive office space to be commuted to. This process ha already started in the field of research, and some establishments such as MIT provide computer terminals for the homes of senior research staff. This is sensible when one considers the tendency for great ideas to materialize in the bath. The future will see such facilities available in less intellectual fields. Many of the better characteristics of the cottage industry may return, particularly in terms of personal freedoms.

Martin & Norman said.

The first widespread use of home terminals will probably be sponsored by employers. Some employees should spend at least part of their time working at home rather than traveling to an office. The overhead cost of providing staff with offices and desks is very high, especially in big cities; a company may cut its costs significantly if most of its personnel work at home. Mothers who participate may be relieved of the boredom they feel when they are unable to leave their children.

We see here a beginning of a return to "cottage industry"— a trend that will probab increase greatly during the next few decades. But at the moment a few factors are counteracting this trend. The first is the reluctance of some companies to give their employees home terminals, because such a step seems an encroachment on leisure time. The second is a feeling that, regardless of what they are doing, employees ought to be at their desks from 9 A.M. to 5 P.M. (At home— who knows— they might be watchi television!) The third is a feeling that people cannot work at home because the environment is not suitable. We believe that these three views are not generally held by intelligent and enthusiastic members of the community, and that this is an era when we must throw off the mores of tradition and rethink what is applicable to the age of teleprocessing.

Kemeny said:

It is worthwhile to analyze the reasons that millions of people rush into central cities each morning. Some of this manpower is necessary to staff the remaining manufacturing installations, but these are rapidly deserting the major cities. Some of it is necessary to maintain the life functions of a city with millions of people. But the major fraction of the daily in-and-out movement is attributable to executives, professionals, and administrative workers who staff offices located in the cities. Why is it necessary for them to make a daily pilgrimage from their homes to the city?

One reason is the desire to talk with colleagues in the office or to counterparts in other offices. An increasing percentage of this talk is now carried on over the telephone. I have heard executives complain that they rush into their offices and then spend half their time talking on the telephone, something they could have done equally well at home.

I predict a dramatic effect on the pattern of employment and location of offices will come about by the widespread use of videotelephones. I myself prefer a face-to-face conversation to a discussion by telephone. However, I feel that with the existence of video telephones conversations and small group discussions at remote locations will be almost as effective as face-to-face discussions.

A second major reason for going to the office is that that is where the files are. However, such information will in the future be kept in national computer networks. Once that occurs, the files of a company will be accessible and updatable from any location in the country.

There is no reason why each executive could not pick a site for his office near his home and recruit secretaries and assistants from the immediate vicinity. The small offices would be linked by means of the national computer networks and the network of video-phones. Such an arrangement could greatly reduce the ridiculous amount of traveling that Americans take for granted. An eight-hour working day could be eight hours plus five minutes travel in the morning and five minutes travel in the evening rather than the eleven-hour day necessitated by the daily battle of getting to the office and coming home. Human beings would then have much more freedom in deciding where and how they want to live their lives.

It is my conviction that if the need for millions of people to rush in and out of the city every working day is removed, we would be well on our way to a solution of urban problems. What exactly would be the role of the central city? Perhaps it will become truly a center for information where the machines are located but not the human beings who use them. It would still need a substantial staff of supporting technicians to look after the machines and the usual services necessary to support the people living in the city, but the density of population could be substantially decreased. Since cities still would have a central location, easily accessible, they might expand their roles as centers for recreation and entertainment. Cities might become important because of the presence of museums, theaters, concert halls, and sport arenas, and as places of living for those who insist on seeing a play or a sports event in person rather than watching it over television.

Martin & Norman:

Instead of coming into a store, the consumer could scan a list of available goods and their prices at several different shops on the home terminal, and then use the terminal to place the order. An organization selling in this way could cut overheads to a minimum by eliminating stores.

The customer who buys through such an automated catalogue avoids the exasperation of fruitless searches for hard-to-find merchandise. Besides replacement bulbs for projectors condemned by planned obsolescence, spare parts for automobiles or rare phonograph records— all items likely to be found, if at all, in only one store in the city— there are tedious searches to be made for unique items like summer houses for rent, theater tickets for a particular show, or boats with a particular specification. Presently, these are found through "classified ads" in newspapers or through agents. Much of the work of such agents, and of the advertising columns, could be more cheaply and efficiently done by a data bank accessible from terminals.

Cuttle:

Most businesses and service organizations will handle day-to-day transactions via terminals, so that the householder can examine his bank statement, scan the latest share prices, buy and sell through his stock computer, and lay bets with his bookie computer. Not merely can he obtain theater, concert, and travel tickets through an agency, but the computer facilities will allow him to interrogate the various possibilities, drawing on a bank of information far greater than any agency today could contain in a brochure or even have readily accessible to its own staff. Agencies will compete not so much in the glossiness of their brochures as in the ease with which their clientele can obtain the information they want from the agency computer.

Kemeny:

My sister says she would love to see home terminals provide easy information on what is available in various stores. She wears rather narrow shoes and often has to go to a dozen different stores before she finds a pair in her size. Businesses could keep their inventory inside the memory of the computer, and any customer in the region could carry out a simple computerized search to find out what is available. The computer would not only record a sale as a financial transaction but would automatically update the inventory of the store. Therefore a clerk or a customer could know immediately what is available, and when certain items are running low the computer could make out an order for replacements.

I am convinced that for items exceeding one dollar in price cash transactions will totally disappear. The multiplicity of credit cards is already a step in that direction. However, popular as today's credit cards are, the system is basically inefficient. You present your credit card in a store and sign a bill. This has to be sent by mail to an office of the credit card company which in turn pays the store. There is enormous waste in all this. As a matter of fact, I suspect that one reason credit cards are so popular is that it often takes as much as two months to receive a bill for a purchase.

A more practical system would be for each person to receive a credit card from his own bank. When buying an item you would present your credit card for insertion in an appropriate computer terminal. This would automatically result in a credit to the account of the store and a debit to your account. If during the transaction you exceed your balance, a red light would come on and the sale would be refused. Or, more likely, the system by which banks provide automatic loans will be expanded, so that moderate overdrafts would result in an interest charge rather than in the refusal of payment.

A pleasant feature of this system would be the fact that it will not be necessary to balance one's checkbooks. This would be done automatically by the bank's computer, and the customer could call up on his home terminal at any time to find out what his balance is.

APPLIANCES

Cuttle:

Anyone who has any doubts about the ability of a computer to cope with breakfast has only to remember the state of mind of the average housewife at seven in the morning to realize that preparing breakfast is a very mechanical task indeed. Many other household tasks are equally suitable for computers to invade; indeed, they have already begun to do so in, for example, washing machines and central heating control systems. At present each piece of equipment needing such a computer has its own small one built in, but the logical development is to have a larger household computer tucked away under the stairs. Circuits could then be wired into the house so that each redundant gadget could be plugged in.

Martin & Norman:

The telephone line, in addition to its normal use, will probably also be used for activating household appliances. A family driving home after a few days away, will telephone their home and then key some digits on the Touch Tone phone which switch on the heat or air-conditioning unit. A woman before leaving for work will preprogram her kitchen equipment to cook a meal. She will then phone at the appropriate time and have the meal prepared.

GOVERNMENT

Cuttle said:

The ability to interrogate the computer is useful in government, both local and national. The householder could readily ask whether any legislation were in progress that affected his neighborhood or interests, and could thus avoid some of the all-too-frequent failures of communication between individuals and local government. The individual could have far easier access to his congressman than the present system permits. Conversely, the householder could be interrogated, and this might well be a highly effective means of keeping congressmen in touch with the feelings of their constituents.

NEWSPAPERS

Kemeny said:

Let us consider a system under which the New York Times, instead of publishing hundreds of thousands of copies, would store the same information in a computer tied into a national network, from which each reader could retrieve the items he wanted, in as much detail as he desired. Each reporter would file his stories by typing them directly into a computer terminal.

A reader sitting in his home could dial into his computer network and ask for his personalized New York Times. The computer would have on file those general topics which the user normally reads. For example, he might normally like to see what is new on the international front, in finance, and in sports. The computer would take these topics up one at a time. In the sports section, it would give the reader a choice of all the different sports covered in that day's newspaper. Let us say that the reader selects baseball and tennis. The computer would list all the available stories under baseball. Each story he chooses would then be presented to him a frame at a time, and he could ask for more details or not as he wished.

I would like to summarize the advantages of this scheme. First, each reader has available to him at any moment, day or night, a completely up-to-date newspaper. It should be perfectly practical to have no more than an hour's lag between the occurrence of a news event and its availability in one's personalized newspaper. Second, the reader would select just those items that are of particular interest to him. This will save him a good deal of time and yet assure him that he does not miss any news that concerns him.

It may be objected that while this scheme sounds great from the reader's point of view, it would bankrupt the Times, which depends on its advertising revenue. But this need not be the case. The computer network would charge the user for various services and could pay a royalty to the Times for each access by a user. Given that the cost of print and distribution is eliminated, the Times may make a profit on "sales" alone. Nor is it necessary to eliminate all advertising. Classified ads could be well indexed and made available to any user who desired to see them. If that did not suffice, the newspaper retrieval program could be written so that between frames it printed ads. However, I would then hope that by paying an extra fee I would have the option of eliminating all advertisements.

I make this suggestion freely available to The New York Times. I suggest that if the newspaper adopts it, it change its motto to "All the news that you see fit to read."

MEDICINE

Cuttle said:

I hope the professionalism of doctors and nurses will not blind them to the possibilities in medicine. Automatic diagnosis by computer could be a useful aid. Interrogation throu a household terminal, with carefully programmed questions, could pin-point some everyday ailments. Where any doubt existed, communication would be transferred to either the G.P. or, if the broad diagnosis were clear, a specialist, presenting him with a neat summary of the facts elicited by the preliminary questioning. Thus the specialist would be spared the routine probing and the G.P. spared the need to be a near-special in everything. The G.P.'s role would, however, remain since in a computer world the personal touch is all the more necessary. Moreover, much treatment can be carried out in the home that today might necessitate hospital treatment. Where observations and tes are required but no emergency is likely to occur, it may be far cheaper and pleasanter for the patient to have monitoring equipment brought to his home and connected throug the terminal to a hospital computer. The latter would interrogate the equipment at intervals, and any change in the patient's condition would be brought to the attention of hospital staff.

THE WHOLE FAMILY

Kemeny said:

Here's how each member of the family might use the computer terminal. Father, if he brings his work home from the office, can use the terminal in place of a sizable office staff. It can have access to the files of his company, it can help him in carrying out research, and at somewhat greater cost it could even transcribe letters and send them for him.

Mother can do most of her shopping through a computer terminal. She can either find out what is available and go in person to make the purchase or actually enter her shopping list in the terminal. The list would be "delivered" to the local supermarke and the goods automatically packaged and loaded on the truck. If by 1990 the roles of man and woman have been completely reversed, the computer terminal will be equally happy to work out business problems for mother and to help father with his shopping and housework.

Children will find the home terminal an immeasurable asset in doing homework. Indeed the child of 1990 will find it impossible to conceive how the older generation managed to get through school without the help of a computer.

All these services can be highly personalized. The computer would be happy to address the user by first name, nickname, or title, once the user's preferences are known. It could store measurements for various items of clothing to aid in computerizec shopping. It could also keep a record of recently performed services so that the compu could "learn" its customer's preferences.

All the applications I have mentioned are possible in principle today. Therefore they should be perfectly practical on a large scale by 1990.

COMPUSERVE

You've been reading the predictions that Martin, Norman, Kennedy, and Cuttle made in 1970. Many of their predictions have come true already. For example, today you can put a terminal in your home and phone a large computer network, such as Compuserve-Micronet.

Here are the 128 services that Compuserve-Micronet offers electronically today:

Home Services

Newspapers: The Columbus Dispatch, The New York Times, Virginian Pilot & Ledger Star, The Washington Post, The San Francisco Chronicle, The San Francisco Examiner, The Los Angeles Times, Minneapolis Star and Tribune, Atlanta Journal Constitution, St. Louis Post-Dispatch, Middlesex Daily News

Weather: Aviation Weather, Public & Marine Weather

Reference Library: Aviation Safety Institute, Better Homes & Gardens, Movie Reviews, Popular Science (Energy), Popular Science (New Products), The Future File, The Refundle Bundle, U.S. Government Publications, Video Information

Communications: Electronic Mail (user-to-user messages), CB Simulation, National Bulletin Board (public messages), User Directory

Shop Bank at Home: Compu*U*Star (Electronic Shop-at-Home), Electronic Banking, Art Gallery & Documentation, New York Times, Minneapolis Star Tribune

Groups and Clubs: AVSIG, CBIG, Hamnet, Netwits, Photo 80, Instructions, Descriptions of Groups

Games and Entertainment: Adventure, New Adventure, Aunt Nettie, Backgammon, Blackjack, Chess, Civil War, Concentration, Craps, Cube Solver, Eliza, Fantasy, Fastermind, Football, Furs, Golf, Gomoku, Hammurabi, Hangman, Lunar Lander, Maze, Mugwump, Othello, Pirates, Roulette, Scramble, Space War, StarTrek, Wumpus

Education: The College Board, The Multiple Choice

Home Management: Balance Your Checkbook, Calculate Your Next Raise, Calculate Your Net Worth, Amortize a Loan

Business and Financial Services

News Reports: CNS Financial Commodity News, Archer Commodity Report, Investment News and Views, Small Business Report, The Federal Report

Reference Databases: MicroQuote, Quick Quote, Standard and Poor's, Value Line Data Base II, Descriptions and Premium Program Rates

Communications: Electronic Mail (user-to-user messages), User Directory

Personal Computing Services

News: Atari Newsletter, RCA Newsletter, Tandy Newsletter, Microsoft Newsletter, The Micro Advisor

Reference: Programming Languages, Text Editors, Word Processing, Utilities, Mathematics and Statistics, Special Features, Terminal Software

Communications: Electronic Mail (user-to-user messages), CB Simulation, National Bulletin Board (public messages), User Directory

Shop at Home: Softex (software exchange program), Access (user-contributed software), Art Gallery & Documentation

Groups and Clubs: CP/M, HUG, LDOS, MAUG, MCCONN, MNET 11, MNET 80, MUSUS, RCA, VTOS, Instructions, Description of Groups

Programming and Computing

Aids

User Information: What's New, Command Summary & Usage Tips, Feedback to Customer Service, Changing Terminal Defaults, Changing Your Password, Reviewing Your Charges, Changing Credit Card Info, Telephone Access Numbers, Current Rates, CompuServe Viewpoint

Index

DANGERS

CHANGE SOCIETY

How could computers change human society? The many good ways are obvious. Here are the bad ones.

ERRORS

Although the computer can have a mechanical breakdown, the usual reason for computer errors is *mental* breakdown—on the part of the people who run it. The usual computer blooper is caused by a programmer who writes an incorrect program, or a user who inputs a wrong number. If you want the computer to write a check for $10.00, but you forget to type the decimal point, the computer will nonchalantly write a check for $1000.

The biggest computer blooper ever made occurred at Cape Kennedy. A rocket rose majestically from its launch pad and headed toward Venus. Suddenly it began to wobble. It had to be destroyed after less than 5 minutes of flight. The loss was put at $18,500,000. What went wrong? After much head-scratching, the answer was finally found. In one of the lines of one of the programs, a programmer omitted a hyphen.

In the computer center of a certain city, every inhabitant's vital statistics were put on cards. One lady in the town was 107, but the number 107 wouldn't fit on the card properly, because the space allotted for AGE was only two digits. The computer just examined the last two digits, which were 07, and assumed she was 7 years old. Since she was 7 and not going to school, the computer printed a truant notice. So city officials visited the home of the 107-year-old lady and demanded to see her mother.

A man in Germany received a bill from a computer requesting the payment of "zero deutschmark". He ignored it, but two weeks later the machine sent him a letter reminding him that he had not paid the sum of "zero deutschmark". Two weeks after that another and more strongly worded letter arrived. He still took no action other than photocopying the letters and gleefully showing them to his friends. But the computer persisted and eventually announced that it was referring his failure to pay to company lawyers. So he telephoned the company. They explained to him there was a minor oversight in the program, assured him it was being corrected, but requested him to send a check for "zero deutschmark" to simplify the reconciliation. He duly made out a check for "0.0 DM." and mailed it. Two days later the check was returned to him from the bank with a polite (nonautomated) letter stating that the bank's computer was unable to process the check.

That last anecdote was from Martin and Norman's *The Computerized Society*. This is from Time Magazine:

Rex Reed, writer and sometime actor, ordered a bed from a Manhattan department store. Three months passed. Then came the long anticipated announcement: the bed will be delivered on Friday. Reed waited all day. No bed. Having disposed of his other bed, he slept on the floor. Next day deliverers brought the bed but could not put it up. No screws. On Monday, men appeared with the screws. But they could not put in the mattresses. No slats. "That's not our department." Reed hired a carpenter to build them; the department store's slats finally arrived 15 weeks later. Undaunted, Reed went to the store to buy sheets. Two men came up and declared: "You're under arrest." Why? "You're using a stolen credit card. Rex Reed is dead." Great confusion. Reed flashed all his identity cards, the detectives apologized—and then tore up his store charge card. Why? "Our computer has been told that you are dead. And we cannot change this."

On a less humorous note, a woman died from freezing, because an errant computer thought she hadn't paid her utility bill.

UNEMPLOYMENT

Since the computer's a labor-saving device, it may make laborers unemployed. Clerks and other low-echelon white-collar workers might find themselves jobless and penniless.

The newspaper companies in New York City have realized they'd save money by hiring fewer printers and using computers instead. But the printers unions, upset, cried "Breach of contract!" The companies and printers finally agreed to get the computers, hire no new printers, but retain the current ones until retirement.

The advent of computers doesn't have to mean a net loss of jobs. In fact, new ones are created. Not all computer related jobs require abstract thinking. There's a need for mechanics, typists, secretaries, salesmen, editors, librarians, etc. There is a need for people to tell the programmers what kind of things to program. Running a computer center is a business, and there's a need for businessmen.

When computers do human work, will there be *enough* work left for us humans to do? Don't worry: when no work is necessary, humans have an amazing talent for inventing it. That's the purpose of Madison Avenue—to create new longings. Instead of significantly shortening the work week, Americans have always opted for a work week of nearly equal length but devoted to more luxurious ends. That's the gung ho Protestant work ethic we're so famous for. Computers will change but not reduce our work.

. . . That's what will happen in the long run. But for the next decade or two, as society shifts to computers, a good many people will be temporarily out of a job.

QUANTIFICATION

Since the computer handles numbers easily, it encourages people to reduce problems to numbers. That's both good and bad. It's good because it forces people to be precise. It's bad because some people are starting to make quantification a goal in itself, forgetting that it's but a tool to other ends. Counting the words that Shakespeare wrote is of no value in itself: it must be put to some use. In both the humanities and the social sciences, I'm afraid the motto of the future will be, "If you can't think, count." Some cynics have remarked, "The problem with computers is that they make meaningless research possible."

Since only quantifiable problems can be computerized, there's a danger that, in a burst of computer enthusiasm, people will decide that unquantifiable problems aren't worth investigating, or that unquantifiable aspects of an otherwise quantifiable problem should be ignored. John Kemeny gives this example:

I've heard a story about the design of a new freeway in the City of Los Angeles. At an open hearing a number of voters complained bitterly that the freeway would go right through the midst of a part of the city heavily populated by blacks and would destroy the spirit of community they'd slowly and painfully built up. The voters' arguments were defeated by the simple statement that, according to an excellent computer, the proposed route was the best possible one. Apparently none of them knew enough to ask how the computer had been instructed to evaluate the variety of possible routes. Was it asked only to consider the costs of building and acquisition of property (in which case it would have found routing through a ghetto area highly advantageous), or was it also asked to take into account the amount of human suffering that a given route would cause? Perhaps the voters would even have agreed it's not possible to measure human suffering in terms of dollars. But if we omit consideration of human suffering, then we're equating its cost to zero, which is certainly the worst of all procedures!

People are being reduced to numbers: telephone numbers, social security numbers, zip codes, etc. When you start treating another human as just a wrong telephone number, and hang up in his face, something is wrong.

ASOCIAL BEHAVIOR

The computer's a seductive toy. When you walk up to it, you expect to spend only a few minutes, but wind up spending several hours instead. Whether catching bugs or playing Pac-Man, you'll probably wile away lots of time. You may find yourself spending more time with the computer than with people. That can be dangerous. For the average American child, his mother is a television set. Will the computer replace T.V. as the national fixation?

Getting along with the computer is easy—perhaps *too* easy. Though it can gripe at you, it can't yell. If you don't like its behavior, you can turn it off. You can't do the same thing to people. Excessive time spent with the computer can leave you unprepared for the ambiguities and tensions of real life.

The computer replaces warmth by precision. Excessive time spent with it might inhibit your development as a loving individual.

IRRESPONSIBILITY

Computerization is part of the oncoming technological bureaucracy. Like all bureaucracy, it encourages the individual to say, "Don't blame me—I can't change the bureaucracy." Only now the words will read, "Don't blame me—the computer did it."

When John Kemeny's sister asked a saleswoman whether a certain item was in stock, the woman said she couldn't answer, because the information was kept by a computer. The woman hadn't been able to answer questions about stock even before the computer came in—the computer was just a new scapegoat.

Computers will eventually be used to run governments and wars. The thought of someone saying, "I can't change that—that's the way the computer does it" is frightening.

CONCENTRATED POWER

As computers amass more and more information about people, the computers will become centers of knowledge. The people who control them—the programmers, sociologists, generals, and politicians—will gain a lot of power. The thought of so much power being concentrated in the hands of a few is frightening. A handful of people, pressing the wrong buttons, could atom-bomb the earth.

Nobody should have complete control over a computer center. The power should be diversified. Sensitive data and programs should be protected by passwords and other devices, so that no single individual can get access to all of it.

CRIME

The computer is the biggest tool in the kit of the white-collar criminal. All he has to do is insert a zero on a tape, and the computer will send him a paycheck for ten times the correct amount. To catch such criminals, computers are programmed to do a lot of double-checking; but if the criminal evades the double-checks, he won't get caught. Police have a hard time tracking down computer criminals, because fingerprints and other traditional forms of evidence are irrelevant. Most computers have passwords, to try to stop people for fooling around with sensitive data, but a bright programmer can devise various tricks to get around the passwords. The crudest is to bug the wires that go to the terminals. The cleverest is to slip extra lines into an innocent program, and get someone else to run it; the extra lines transfer money to the programmer's account.

Since you must be quite smart to be a computer criminal, even if you're caught you are likely to be admired. The usual reaction of people is not—"What a terrible thing you've done!"—but rather—"Gee, you must be smart. Tell me, how did you do it?" A bright button-down computer criminal who steals $100,000 electronically usually gets a lighter sentence than the dude who must resort to a gun to get $1000. Is that justice?

INVADED PRIVACY

Of all the harm computers can do, "invaded privacy" worries people the most. George Orwell, in his book *1984*, warned that someday "Big Brother will be watching you" via a computer. His prediction's already a reality: your whereabouts are constantly checked by computers owned by the FBI, the IRS, the military, credit-card companies, and mail-order houses. My brother once wrote an innocent letter asking for stamps. Instead of using his own name, he used the name of our dog, Rusty. Since then, we've received letters from many organizations, all addressed to "Mr. Rusty". Our dog's name sits in computers all across the country.

The information computers have stored about you may be misleading. If you never find out about the error, the consequences can haunt you the rest of your life. Examples:

A teacher saw one of the little boys in her class kiss another boy. She entered on his computerized school records, "displays homosexual tendencies".

According to computer records, a certain man had "three lawsuits against him". In fact, the first was a scare suit 30 years before, over a magazine subscription he had never ordered; the second had been withdrawn after a compromise over a disputed fee; the third case had been settled in his favor.

You have a right to see what information is stored about you, and change it if it's incorrect. For example, if a teacher or employer writes a "confidential recommendation" about you, you have a right to examine it, to prevent misleading statements from haunting you for life.

Even if the information stored about you is accurate, you have a right to prevent its dissemination to the world in general. No organization should store or disseminate information unjustifiably.

What is "justifiable"? Fearing "Big Brother", people don't want politicians to access personal information. On the other hand, fearing criminals, people want the police to have a free hand in sleuthing. How to give information to the police without giving it to politicians can be puzzling.

Outdated information should be obliterated. An individual shouldn't be haunted by his distant past; he should be given a chance to turn over a new leaf. Moreover, information 50 years old may be couched in words that have been redefined. To be a "leftist", for example, means something in each decade.

Only facts should be stored, not opinions. It's okay to store that someone lives on Fifth Avenue, but not that he lives in a "nice neighborhood".

It's unfortunate that people feel a need for privacy. If the information stored about you is correct, why argue? But many people feel a need to be secretive, and I suppose people do have that right. It's sometimes called the right to be "let alone".

People don't want to feel their whole lives are on stage, recorded by a computer. It inhibits them from acting free and natural. Even if the computer doesn't store any damaging information about you, the mere *thought* that your every action is being recorded is damaging, because it makes you act more conservatively. You may be afraid of adopting a good but unusual lifestyle, because anything "different" about you will look bad on the computerized records used by banks, credit-card companies, insurance companies, and other conservative institutions. The harmful thing is not that Big Brother is watching, but that you *feel* he's watching. You are subjugated.

JOBS

PROGRAMMER

A *programmer* is a teacher: the programmer teaches the computer new tricks. For example, the programmer might teach the computer how to do the payroll. To do that, the programmer feeds the computer a list of instructions, that explain to the computer how to do the payroll. The list of instructions is called a *program*.

Languages

The program is written by using the very limited vocabulary that the computer understands already. In volumes 1 and 2, I explained a vocabulary called BASIC, which consists of words such as PRINT, INPUT, GO TO, IF, THEN, and STOP. That vocabulary— BASIC— is called a *computer language*. It's a small part of English. No computer understands the whole English language. The programmer's job is to translate an English sentence (such as "do the payroll") into language the computer understands (such as BASIC). So the programmer is a translator.

Some computers understand BASIC. Other computers understand a different vocabulary, called COBOL. For example, COBOL uses the words DISPLAY and WRITE instead of PRINT.

Before programming a computer, you must find out which language the computer understands. Does it understand BASIC? Or does it understand COBOL instead? Or does it understand a yet different language? The most popular languages are BASIC, COBOL, FORTRAN, and PASCAL; but there are also *thousands* of others. Your computer understands at least one of those languages; if you're lucky, your computer understands *several* of those languages.

When you apply for a programming job, the first question to ask the interviewer is which languages does the company's computer understand? Or better yet, ask, "Which language do you want me to program in?" The interviewer will say "BASIC" or "COBOL" or some similar answer and then ask you, "Do you know that language?"

Most microcomputers use BASIC. Most maxicomputers use COBOL for applications to business, and use FORTRAN for applications to engineering. Minicomputers often use BASIC, but sometimes use COBOL or FORTRAN. The language PASCAL, which is newer, isn't used as much as BASIC, COBOL, or FORTRAN yet, but its popular is gradually increasing.

Of the four popular languages, the easiest is BASIC. It's also the most fun. To become a programmer, begin by studying BASIC, then move on to the other languages which are yukkier.

Because BASIC is so easy, saying you know "BASIC" is less prestigious than saying you know "FORTRAN" or "COBOL" or "PASCAL". To get lots of prestige, learn *many* languages. Even if the job you're applying for requires only one language, you should say that you know many languages well, to convince the interviewer that you're brilli

The most prestigious languages to know are "assembly language" and "machine language", because they're the most difficult. If you can convince the interviewer that you know assembly language and machine language, the interviewer will assume you're God and offer you a very high salary, even if the job doesn't require a knowledge of those languages.

Specific computers

Before going to the interview, learn about the specific computer the company uses. For example, if the company's computer is an IBM 370, study the IBM 370's details. Study its operating system and its languages. If the job requires COBOL, study the particular dialect of COBOL used on the IBM 370. Each computer has its own dialect of COBOL, its own dialect of BASIC, its own dialect of FORTRAN, its own dialect of PASCAL. Usually, the differences between dialects are small, but you must know the For "assembly language" and "machine language", the differences between dialects ar much greater: the "assembly language" or a Radio Shack computer is almost entirely different from the "assembly language" on an IBM 370.

Analysis versus coding

The act of programming consists of two stages. The first stage is to analyze the problem, to make the problem more specific. For example, suppose the problem is, "Program the computer to do the payroll". The first stage is to decide exactly how the company wants the payroll to be done. Should it be done weekly, bi-weekly, semi-monthly, or monthly? Should the records be stored on disks, or on tapes instead? (Disks are nicer, but more expensive.) While computing the payroll checks, what many reports do you want the computer to generate? For example, do you want the computer to also print a report about the employees' attendance, and about how much money each department of the company is spending on salaries? Maybe one of the departments is over-budgeting. And what kind of paychecks do you want the computer to refuse to print? For example, if somebody in the company tries to make the computer print a paycheck for a ridiculous amount (such as $1,000,000 or $?), you want the computer to refuse (and perhaps signal an alarm).

That stage— analyzing a vague problem (such as "do the payroll") to make it more specific— is called *analysis*. A person who analyzes is called an *analyst* or, more prestigiously, a *systems analyst*. After analyzing the vague problem and transforming it into a series of smaller, more specific tasks, the analyst turns the problem over to a team of *coders*. Each coder takes one of the tasks and translates it into BASIC or COBOL or some other language.

If you're hired to be a "programmer", your first assignment will probably be as a coder. After you gain experience, you'll be promoted to a systems analyst.

The ideal systems analyst knows how to analyze a problem but also has had prior experience as a coder. A systems analyst who knows how to both code and analyze is called a *programmer/analyst*. An analyst who doesn't know how to code— who doesn't know BASIC or COBOL— who merely knows how to break a big problem into a series of little ones— is paid less.

Three kinds of programming

Programming falls into three categories: *development*, *testing*, and *maintenance*. *Development* means inventing a new program. *Testing* means making sure the program works. *Maintenance* means making minor improvements to programs that were written long ago. (The "improvements" consist of eliminating errors that were discovered recently, or making the program conform to changed government regulations, or adding extra features so that the program produces extra reports or handles extra special cases.) Development is more exciting than testing, which is more exciting than maintenance. So if you're a new programmer, the other programmers will probably "stick you" in the maintenance department. You'll be part of the maintenance crew. Since your job will consist of "cleaning up" old programs, cruel programmers will call you a "computer janitor".

"Application program" versus a "systems program"

Programs fall into two categories. The usual kind of program is called an *application program*; it handles a specific application (such as "payroll" or "chess" or "send rocket to moon"). The other kind of program is called a *systems program*; its only purpose is to help programmers write applications programs.

For example, hidden inside the computer is a program that makes the computer understand BASIC; that program explains to the computer what the words PRINT, INPUT, GO TO, IF, THEN, and STOP mean. That program (which is called the *BASIC language processor*) is an example of a systems program.

Another systems program is called the *operating system*. It tells the computer how to control the disks and printer and terminals. If the operating system is fancy, it even tells the computer how to handle many programmers at once.

Another systems program, found on large computers, is the *editor*. It lets you edit programs written in languages such as COBOL and FORTRAN.

So systems programs are tools, which help programmers write application programs. When you buy a computer, buy some systems programs, so you can create applications programs easily.

A person who invents systems programs is called a *systems programmer*. To become a systems programmer, learn assembly language and machine language.

Creating a systems program is very difficult; so a systems programmer usual gets paid more than an applications programmer.

The word "systems" is prestigious: it's used in the phrase "systems analyst" and in "systems programmer". In some companies, if your boss wants to praise you, the boss will put the word "systems" in front of your title, even if your job has nothing to do with "systems".

How to learn programming

To be a good programmer, you need experience. You can't become a good programmer by just reading books and listening to lectures; you must get your hands on a comput and practice.

If you take a computer course, the books and lectures are much less valuable than the experience of using the school's computer. Spend lots of time in the computer cent Think of the course as just an excuse to get permission to use the school's computer. The quality of the lecturer is less important than the quality of the school's computer center. The ideal computer center uses video terminals instead of punched cards; has a computer that can understand many languages; gives you *unlimited* use of the computer (no "extra charges"); is open 24 hours a day; has enough terminals so you don't have to wait in line for somebody else to finish; has a staff of "teaching assistan who will answer your questions; has a rack full of easy-to-read manuals that explain h to use the computer; lets you borrow books and manuals, to take home with you; and has *several* kinds of computers, so that you get a broad range of experience. Before you enroll in a computer course, find out whether the school's computer center has th features.

Many computer schools are unnecessarily expensive. To save money, take fewer courses, and buy more books and magazines instead. Better yet, buy a computer yourself, and keep it in your home! You can buy a VIC for just $85; it's a nice computer, excellent for learning BASIC. Keep the VIC for a few months, learn all you can from it, then sell it to a friend for $45— so that using the VIC has cost you just $40 and given you several *months* of education. That's a much better investment of your money than spending many hundreds of dollars for a computer course.

Another cheap way to get an education is to phone your town's board of education, and ask whether the town offers any adult-education courses in computers. Some town offer adult-education computer courses for under $100.

For an even better deal, phone your town's board of education— or high school— and ask whether you can sit in the back of a high-school computer class. If you're an adult resident of the town, you might be able to sit in the back of the class for free. Your only "expense" will be the embarrassment of sitting in the same room as youngst After a day or two of feeling strange, you'll get used to it, and you'll get an excellen education, free.

Community colleges offer low-cost courses that are decent. Explore the community colleges before sinking money into more expensive institutions that are over-priced.

Starting salary

For your first programming job, your salary will be somewhere between $12,000 and $24,000. The exact amount depends on which languages you know, how many program you wrote previously, whether you have a college degree, whether you've had experience on the particular kind of computer the company uses, and whether you know the application area. (For example, if you're a programmer for an insurance company, it's helpful to know something about insurance.)

Degrees

A college degree is not essential, but helpful. Try to get a degree in "computer science" or "management information systems". "Computer science" emphasizes the underlying theory, systems programming, assembly language, PASCAL, FORTRAN, and applications to science; "management information systems" emphasizes BASIC, COBOL, and applications to business. A major in "mathematics" that emphasizes computers is also acceptable.

Discrimination

If you're a woman or non-white or physically handicapped, you'll be pleased to know that the computer industry discriminates less than in other occupations. In fact, being a woman or non-white or physically handicapped works to your *advantage*, since many companies have affirmative-action programs.

On the other hand, discrimination *does* exist against older people. If you're over 40 and trying to get a job as an entry-level programmer, you'll have a tough time, since the stereotypic programmer is "young, bright, and a fast thinker". If you're old, they'll assume you're "slow and sluggish".

Because of that unfair discrimination, if you're old you should probably try entering the computer industry through a different door: as a consultant, or a computer salesperson, or a computer-center manager, or a computer teacher. For those positions, your older age works to your *advantage*, since those jobs require *wisdom*, and people will assume that since you're old, you're wise.

Shifting careers

If you're older, the best way to enter the computer field is to combine your new knowledge of computers with what you knew previously. If you already knew a lot about sales, get a job selling computers. If you already knew a lot about teaching, get a job teaching about computers— or helping teachers deal with computers. If you already knew a lot about real estate, computerize your real estate office.

In other words, do *not* try to "hop" careers; instead, gradually *shift* your responsibilities so that they deal more with computers.

To get into the computer field safely, retain your current job but computerize it. For example, if you're already a math teacher, keep teaching math but also convince your school to let you teach a computer course also, or at least incorporate computers into the math curriculum, or at least help run the school's computer center. If you already work for a big company and are doing a boring job, try to get transferred to a different department and which puts you in closer contact with the computer. After a year in such a transitional state, you can break into the computer field more easily, since you can put the word "computer" somewhere on your resumé, as "job experience".

If you're in college kid, write programs that help the professors, or help others during your summer vacations. Agree to write the programs for little or no pay. Your goal is *not* money: your goal is to put "experienced programmer" on your resumé.

Interviews

When applying for your first computer job, avoid the "personnel" office. The people in the personnel office will look at your resumé, see it includes too little experience, and trash it.

Instead, play the who-you-know game. Contact somebody who actually works with computers, convince that person you're brighter than your resumé indicates, and prove to that person you've learned so much (from reading, courses, and practice) that you can conquer any task laid before you. Convince that person you can think quickly. If you impress that person enough, that person can get you a job, even though your paper qualifications look too brief.

When you get an interview, be aggressive: assert yourself. Ask the interviewer more questions than the interviewer asks you. Ask the interviewer about the company's computer, and about why the company doesn't have a different one instead. Ask the interviewer how the other people in the company feel about the computer center. Ask the same kinds of questions a data-processing manager would ask. That way, the interviewer will assume you have the potential to become a data-processing manager, and will hire you immediately. You'll also be showing you *care* enough about the company to ask questions. And you'll be showing you have a vibrant personality, and are not just "another vegetable who came through the door".

One of the strange things about applying for a programming job is that the interviewer will *not* ask to see a sample of your work. The interviewer doesn't have time to read your program. Even if the interviewer *did* have time to read your program, he couldn't be sure you wrote it yourself. Instead, the interviewer will just *chat* with you about your accomplishments. So you must "talk smart". The best way to "talk smart" is to know all the buzzwords of the computer industry— even if they don't really help you write programs.

During the interview, you'll probably be asked whether you know "structured programming". A *structured program* is a program that's well-organized. It consists of a short main routine and many subroutines. To write a structured program, avoid the words GO TO; instead, use words that involve subroutines:

Language	Words that involve subroutines
BASIC	GO SUB and RETURN
FORTRAN	CALL, SUBROUTINE, and RETURN
PASCAL	PROCEDURE
COBOL	PERFORM

Later joys

In your first job, your salary will be low, but don't worry about it. During your first job, you'll receive lots of training: you're getting a free education. After your training period is over, your salary will rise rapidly— especially if you do extra studying during evenings and weekends. Your *real* job is: to become brilliant.

After you've become brilliant and experienced, other companies will eagerly want to hire you. Your best strategy is to leave your current company and work elsewhere, to gain new experiences. Whenever you feel you're "coasting", and not learning anything new, it's time to move to a different job. The "different job" can be in a new company or in a different department of the current company.

By moving around— by gaining a wide variety of experiences— you can eventually become a qualified, wise consultant. And you'll feel like "King of the World".

Social contacts

Being a programmer is not always glamorous. You'll spend many long hours staring at your terminal's screen and wondering why your program doesn't work. The job is intellectual, not social. But after you've become an expert coder, you get into "systems analysis" and "consulting" and "teaching" and "management", and interact with more people.

Software publishing

To be a programmer, you do *not* have to work for a large company. Instead, you can sit at home, write programs on your personal microcomputer, and sell them to software publishers, for a royalty. If the software publisher sells many copies of your program, you become rich. (On the other hand, if your program is not a "smash hit", you remain poor.)

Since your program might not become popular, do not rely on software publishing as a steady source of income. Instead, view it as a part-time activity which, if successful, will put some extra money in your pocket.

The most famous software publishers are Instant Software, Hayden, Broderbund, Sierra On-Line, Spinnaker, and Microsoft; and there are many others. Browse through the ads in microcomputer magazines.

Software houses

A company whose only goal is to produce software is called a *software house*. You can work for a software house full time (for a fixed salary) or for a royalty. If the software house deals with large computers, the usual method of payment is full-time fixed-salary, rather than royalty.

Programming is fun for young kids. But as you get older, you'll tire of machines, and want to deal with people instead. As you approach retirement, you'll want to help the younger generation relate to the computers you've mastered.

To be a successful manager, you need three skills: you must be technically competent; you must be wise; and you must know how to handle people.

You should know how to program. You should know the strengths and weaknesses of each computer company, and be able to compare their products. You should have a philosophy about what makes a "good" computer center. You should understand people's motives, and channel them into constructive avenues. You should keep up-to-date, by reading the latest books and periodicals about computers, and by chatting with other computer experts via phone and at conventions. If you live in New England and care about microcomputers, join the Boston Computer Society (1 Center Plaza, Boston, MA 02108, 617-367-8080).

When trying to run a computer center, you can easily make mistakes. For example, many computer centers put four-foot-high partitions between their programmers, to give the programmers "privacy"; unfortunately, the partitions are counter-productive they're too low to block noise, and too high to permit helpful conversation with your neighbor.

If you're putting a computer center into a high school (or junior high or middle school or even an elementary school), you must develop a *cadre* of hot-shot students who are bright, friendly, and outgoing, and who will help and encourage the other students to use the computer. If the hot-shots are *not* outgoing— if they become an elitist, snobbish club— the rest of the school will avoid the computer.

If you've hired "programming assistants" who help the programmers, don't let the "programming assistants" hide in an office or behind a desk; tell the programming assistants to go to the terminals, chat with the programmers, and ask the programmer whether the programmers could use any help.

In too many organizations, terminals are locked in the offices of prestigious people and aren't used. Let *everybody* share the terminals.

Too often, managers judge their own worth by the size of the computer center's budget: the bigger the budget, the more prestigious the manager. Remember that the sign of being a good manager is *not* having a big budget; the sign of a good manager is the ability to meet the company's needs on a *small* budget.

In computer jargon, a *word* is how many bits the CPU can handle simultaneously. For most microcomputers, a word is 8 bits (i.e., the CPU can handle 8 bits simultaneously); for most minicomputers, a word is 16 bits; for most maxicomputers, a word is 32 bits. A CPU having a big word is prestigious, but might not be cost-effective.

Too often, the head of the computer center decides who can use the computer. So the head of the computer center becomes powerful— and evil. To avoid concentrating so much power in the hands of one bureaucrat, use *distributed processing*: get sever small computers instead of one big monster, and give each department its own small computer.

If you're a "microcomputer consultant" and honest, you'll tell your client to buy low-cost popular programs, instead of telling him to pay you to invent "customized" programs.

SALES

You can find three kinds of salesmen.

The "slick" kind knows "how to sell", but doesn't know any technical details about the computer he's selling; he doesn't know how to program, and doesn't know much about the computers sold by his competitors. All he knows is the "line" that his boss told him to give the customers. That kind of salesman usually resorts to dishonest tactics such as claiming that all computers sold by competitors are "toys".

The opposite kind of salesman is technical: he knows every detail about every computer manufactured, but can't give you any *real* advice about which computer best meets *your* needs.

The best kind of salesman is a consultant: he asks lots of questions about your particular needs, tells you which of his computers meets your needs best, and even tells you the *limitations* of his computer and why another, more expensive computer sold by a competitor might be better. He's an "honest Joe". He clinches the sale because you trust him, and because you know you won't have any unpleasant surprises after the sale. While selling you a computer, he teaches you a lot. He's a true friend.

A woman can sell computers more easily than a man. That's because most computer customers are men, and men are more attracted to women. It's also because, in our society, women are more "trusted" than men. But if you're a woman, say some technical buzzwords, to convince the customer that you're technically competent; otherwise, the customer will assume that since you're a woman you must be a "dumb secretary".

ORIGINAL JOBS

How about starting a rental service, where people can rent microcomputers? How about starting a camp, where kids can spend the summer playing with computers? How about starting a computer set-up service, where you and your students help executives get started using their Apple and Radio Shack computers? How about writing easy-to-use explanations about how to select and use the most popular computers? Each of those ideas has been tried successfully; join the fun!

WHAT TO READ

PERIODICALS

The best way to learn what's happening in the computer industry is to read magazines and newspapers. The major computer magazines and newspapers are owned by three conglomerates: CW Communications, Ziff-Davis, and McGraw-Hill.

The best magazines about microcomputers are *Infoworld* (published by CW Communications), *Creative Computing* (published by Ziff-Davis), *Popular Computing* (published by McGraw-Hill), and *Byte* (also published by McGraw-Hill). You should subscribe to all four of those magazines.

Infoworld comes out weekly; the others come monthly. Infoworld emphasizes hot news, industry gossip, and software reviews. Creative Computing emphasizes arts, games, education, and the least expensive microcomputers for use at home. Popular Computing emphasizes tutorials for beginners and comparative surveys of competing computers and software. Byte emphasizes computer languages, electronics, and in-depth reviews of new computers.

Infoworld is the closest to being a newspaper. Creative Computing is the most light-hearted. Popular Computing is the easiest. Byte is the most advanced.

A one-year subscription costs $31 for Infoworld, $25 for Creative Computing, $15 for Popular Computing, and $21 for Byte. So altogether, you'll spend $92 for all four.

Infoworld is at 617-879-0700 or 800-343-6474. Creative Computing is at 201-540-0445 or 800-631-8112. Popular Computing and Byte are at 603-924-9281.

Those magazines all deal with microcomputers. For information about larger computers, get *Computerworld*. It's a weekly newspaper that costs $44/year from CW Communications at 617-879-0700.

All those publications come out monthly or weekly. To find out what's happening *today* in the computer industry, read *The Wall Street Journal*, which comes out every day. It's a newspaper that covers American business in general and also includes in-depth articles about each day's developments in the computer industry. Whenever an important new development occurs in the microcomputer industry, it's reported the next day in The Wall Street Journal, then reported the next week in Infoworld, then reported the next month in the other magazines (Creative Computing, Popular Computing, and Byte). The Wall Street Journal costs $94/year; its editorial offices are at 22 Cortlandt St., New York, NY 10007.

Certain computers have their own magazines.

For the IBM PC computer, the most popular magazines are *PC Magazine* and *PC World*. They are both monthly. PC Magazine costs $35/year from Ziff-Davis at 212-725-4694; PC World costs $24/year from CW Communications at 415-861-3861.

For Radio Shack computers, the fattest magazines (which have the most ads for discount dealers and independent sources of hardware and software) are *80 Micro* and *The Rainbow*. 80 Micro costs $36/year from CW Communications at 603-924-9471; The Rainbow costs $22/year from Falsoft at 502-228-4492. The Rainbow covers the Color Computer; 80 Micro covers larger computers.

For teachers, the main magazines are *Classroom Computer News* and *Electronic Learning*. Classroom Computer News costs $16/year from Intentional Educations at 341 Mt. Auburn St., Watertown, MA 02172. Electronic Learning costs $19/year from Scholastic, whose editorial offices are at 730 Broadway, New York, NY 10003.

WHERE TO GET BOOKS

To find good books about computers, visit your local computer store, bookstore, library, and college. If your local bookstore doesn't have the book you're looking for, the bookstore's manager will gladly order it for you.

In New York City, the largest collection of computer books is at the McGraw-Hill Bookstore (1221 Avenue of the Americas, New York, NY 10020, phone 212-997-1221). In Los Angeles, the largest collection of computer books is at the Computer Bookstore (1033 N. Sycamore Ave., Los Angeles, CA 20038, phone 213-464-4322). In other parts of the country, the easiest way to find computer books is to walk into your local branch of B. Dalton Bookseller. If you live near Boston, try B. Dalton (in Boston's Prudential Center), the Tech Coop (which is MIT's bookstore), and Harvest Computer Store (in Cambridge at 576-0600).

BEST BOOKS

To learn about computers, begin by reading *The Secret Guide to Computers*, volumes 1 and 2.

Then read the manuals that came with your computer. Some of those manuals explain the computer's hardware; others explain the computer's software. Although those manuals are hard for a beginner to understand, you can understand them after you've mastered *The Secret Guide to Computers*.

After reading The Secret Guide to Computers and the manuals that came with your computer, read some of the books listed below. For each topic, I've listed the two best books. If you read both books about the topic, you'll become an expert.

For each book, I've listed the title, author, price, and publisher. I've rounded the price to the nearest dollar. But when you buy the book, the price will probably be higher anyway: publishers raise their prices every few months.

BOOKS FOR BEGINNERS

Introductory surveys that include many photographs
The Personal Computer Handbook by Rodwell ($15 Barron's)
Introduction to Computers & Data Processing by Shelly & Cashman ($28 Anaheim)

BASIC
BASIC & the Personal Computer by Dwyer & Critchfield ($16 Addison-Wesley)
A Guide to Programming—IBM Personal Computer by Presley ($17 Lawrenceville)

Amusing facts about the computer subculture
The Naked Computer by Rochester & Gantz ($16 Morrow)
The Hacker's Dictionary by Steele et al ($6 Harper & Row)

Peculiarities of Apple programming
Apple 2 User's Guide by Poole et al ($18 Osborne/McGraw-Hill)
Apple 2 Computer Graphics by Williams et al ($20 Brady)

Pecularities of programming low-cost computers
A Guide to Programming the Commodore Computers by Presley ($16 Lawrenceville)
Your Atari Computer by Poole et al ($18 Osborne/McGraw-Hill)

BOOKS ABOUT ALTERNATIVE COMPUTER LANGUAGES

FORTRAN
FORTRAN 77 for Humans by Page & Didday ($19 West)
The Elements of FORTRAN Style by Kreitzberg & Shneiderman ($10 Harcourt BJ)

PASCAL
Introduction to PASCAL by Zaks ($17 Sybex)
Oh! PASCAL! by Cooper & Clancy ($18 Norton)

COBOL
A Simplified Guide to Structured COBOL Programming by McCracken ($22 Wiley)
Structured ANS COBOL by Murach & Noll (2 volumes, $20 each, Mike Murach)

PL/I
PL/I Programming with PL/C by Davidson ($17 Houghton Mifflin)
PL/I Structured Programming by Hughes ($34 Wiley)

LOGO
LOGO for the Apple by Abelson ($16 McGraw-Hill)
Mindstorms by Papert ($7 Basic Books)

FORTH
Starting FORTH by Brodie ($18 Prentice-Hall)
FORTH Fundamentals Volume 1 by McCabe ($16 Dilithium)

C
The C Programming Language by Kernighan & Ritchie ($20 Prentice-Hall)
The C Primer by Hancock & Krieger ($15 McGraw-Hill)

ADA
Programming in ADA by Wegner ($26 Prentice-Hall)
ADA an Introduction by Ledgard ($19 Springer-Verlag)

DBASE 2
Everyman's Database Primer by Byers ($15 Ashton-Tate)
DBASE 2 User's Guide by Green ($29 Softwarebanc)

Surveys of classical computer languages
Introduction to Programming Languages by Peterson ($27 Prentice-Hall)
Programming Languages by Tucker ($36 McGraw-Hill)

Languages for handling strings
LISP by Winston & Horn ($19 Addison-Wesley)
The SNOBOL4 Programming Language by Griswold et al ($18 Prentice-Hall)

Languages for handling numbers
Statistical Package for the Social Sciences by Nie et al ($17 McGraw-Hill)
APL an Interactive Approach by Gilman & Rose ($24 Wiley)

BOOKS ABOUT OPERATING SYSTEMS

Many microcomputers use the CP/M operating system. The best book about CP/M is *CP/M and the Personal Computer* by Dwyer & Critchfield ($20 Addison-Wesley). That book covers CP/M and also everything that's related to CP/M: it tutors you in Microsoft BASIC, Wordstar, Supercalc, Dbase2, the Peachtree Accounting System, the C programming language, assembly language, and much more! The explanations are fun, though somewhat rambling. Besides that book, you'd also enjoy reading a more focused explanation of CP/M itself, namely the *Osborne CP/M User Guide* by Hogan ($16 Osborne/McGraw-Hill).

IBM maxicomputers use an operating system that involves JCL. The best book about JCL is *System/370 Job Control Language* by Brown ($23 Wiley).

BOOKS ABOUT DATA-BASE MANAGEMENT SYSTEMS

To learn the underlying theory, read *Database Processing* by Kroenke ($32 Science Research Associates); it also compares the popular DBMS systems for maxicomputers. To compare the popular DBMS systems for *micro*computers, read *Data Base Management Systems* by Kruglinski ($17 Osborne/McGraw-Hill).

BOOKS ABOUT ADVANCED PROGRAMMING TECHNIQUES

To learn about 3-dimensional graphics, read *Principles of Interactive Computer Graphics* by Newman & Sproull ($36 McGraw-Hill). To make the computer solve equations and do calculus, read *Introduction to Numerical Methods* by Stark ($28 Macmillan) and then read *Elementary Numerical Analysis* by Conte & deBoor ($29 McGraw-Hill).

BOOKS ABOUT ASSEMBLY LANGUAGES

These books explain the most popular assembly languages for microcomputers:

6502 Assembly Language Programming by Leventhal ($19 Osborne/McGraw-Hill)
Z-80 Assembly Language Programming by Leventhal ($19 Osborne/McGraw-Hill)
The 8086/8088 Primer by Morse ($15 Hayden)

The assembly language for IBM maxicomputers is called *BAL*. To learn BAL, get *Assembler Language Programming* by Tuggle ($20 Solomon Research Associates). To learn assembly language for PDP-11 minicomputers, get *Computer Organization & Programming* by Gear ($34 McGraw-Hill).

To learn how to use BAL to create operating systems and your own computer languages, read *Systems Programming* by Donovan ($38 McGraw-Hill), and then read *Operating Systems* by Madnick & Donovan ($38 McGraw-Hill).

GET PERSONAL

PHONE ME

Whenever you have a question about computers, phone me at 617-266-8128, and I'll help you, free, even if your question is weird.

The kinds of questions I get asked depend on the season of the year, the phase of the moon, and the lunatics who read this book. To give you a taste of what my life is like, I took notes on the calls that I received on or about January 1, 1984.

The notes show that most of the calls were from people who had bought defective Adams, or who couldn't figure out how to make their Adams work. Some calls were about how to buy The Secret Guide to Computers. The remaining calls were the following, which I've listed in chronological order:

Is there an alternative to using the IBM PC monochrome card?
What's the best way to get trained in using computers?
How do you convert "MID$=" to Commodore computers?
Is buying timesharing a good idea?
Which software companies are the biggest?
Why doesn't the Timex Sinclair 2068 accept brackets instead of parentheses?
How do you test for end of data, in Texas Instruments 99/4A BASIC?
How can IBM Personal Computers be shipped to Saudi Arabia?
Where can a good instructor be found, for teaching the Lotus 1-2-3 spreadsheet?
How do you make "DEF FN" and "LINE INPUT" work, on a diskless TRS-80 model 1?
Who publishes a dictionary of CAD/CAM terms?
Which computer is the best for graphics?
On the Radio Shack Color Computer, why does the PRINT statement do wordwrap?
How do you create a data file on a TRS-80 model 3 tape?
Who sells up-to-date reports about the personal computer industry?
Is the Kaypro 10 a good computer to buy, and which printer is best?
How can CBS News find out about people who have discarded personal computers?
Which brands of monitors and cables work with the Texas Instruments 99/4A?
How can the signal that the Commodore 64 sends to the TV be improved?
Who's selling an Osborne 1 computer for $600?
Can the TI 99/4A computer be modified, to yield more characters per line?
Why doesn't the Apple 2e permit a space before the "A" in "COPYA"?
Is the Texas Instruments Professional Computer a good buy?
Which terminal is best?
Why does the Color Computer say "READ ERROR" if you leave the tape on the TV?
Does the DEC Rainbow's price include BASIC?
How do you attach the U.S. Robotics modem to a Morrow MD-3 computer?
Should the owner of a Timex-Sinclair 2068 computer buy Timex's tape recorder?
Why are so many Commodore 64's defective?
Why is the IBM PC's BASIC so buggy, and will "NEWDOS" cure the bugs?
Does the price of the IBM PC Junior include BASIC?
In Microsoft PASCAL, what replaces INKEY, and how do you trap system errors?
What's the best computerized rhythm box, and which computer does it attach to?
Which computer is the best for PASCAL and C?
Where can Ovation find an employee to write a printer driver and convert files?
How can you write a 200 char./line word processor in PASCAL for the Apple 3?
Where can writers be found, to create disk-based computer tutorials?
Of the computers that imitate the IBM PC, which is the best buy?
Which is better: a Commodore 64 or an Apple 2e?
To activate the Color Computer's printer, why must you say PRINT#-2?
Is the Lobo computer worthwhile?
Which computer is best for word processing?
Which computer companies would want to locate in a new industrial park?
What's the fastest way to get a job as a technical writer?
How do companies that rent software protect against piracy?
For word processing, which is better: a Morrow computer or a Radio Shack 4P?
Which is better: a DEC Rainbow or an IBM PC?

For each of those questions, I either gave the answer or told the person how to figure out the answer himself.

If you know the answer to most of those questions already, phone me. You're welcome to have my job. I could use a rest!

BLITZ COURSES

I hope you've enjoyed The Secret Guide to Computers. If you practice what you've read, you'll become a computer expert. When you do, your friends will ask you to teach courses. The following schedules will help you design power-packed high-speed fun bouncy "blitz" courses that summarize the entire computer industry quickly:

One-day blitz course (4 sessions, 2½ hours each)
Morning: the computer industry & personal programming (volume 1 chapters 1-3)
Early afternoon: hardware & popular programming (vol. 1 ch. 3-4, vol. 2 ch. 1)
Late afternoon: applications (volume 1 chapter 5, volume 2 chapter 2)
Evening: climax (volume 1 chapters 6-7, volume 2 chapters 3-10)

Weekend blitz course (4 sessions, 3½ hours each)
Saturday morning: the industry, programming, & hardware (volume 1 chap. 1-3)
Saturday afternoon: pop programming & applications (v. 1 ch. 4-5, v. 2 ch. 1-2)
Sunday morning: advanced techniques (volume 1 chap. 6-7, volume 2 chap. 3)
Sunday afternoon: languages & lives (volume 2 chapters 4-10)

One-week blitz course (5 full days)
Monday: the industry, personal programming, & hardware (volume 1 chapters 1-3)
Tuesday: popular programming & microcomputers (volume 1 ch. 4, volume 2 ch. 1)
Wednesday: applications (volume 1 chapter 5, volume 2 chapter 2)
Thursday: advanced techniques (volume 1 chapters 6-7, volume 2 chapter 3)
Friday: computer languages & lives (volume 2 chapters 4-10)

Three-week blitz course (3 full weeks)
First week: fundamentals (volume 1 chapters 1-4, volume 2 chapter 1)
Second week: applications & professional programs (v. 1 ch. 5-6, v. 2 ch. 2-3)
Third week: arts, brains, languages, & lives (volume 1 ch. 7, volume 2 ch. 4-10)

In the one-day blitz course, each topic is covered superficially; in the three-week blitz course, each topic is covered in depth; the other courses are intermediate.

When giving a blitz course, don't try to cover every detail: you won't have enough time. Tell the audience that the details can be found in The Secret Guide to Computers.

Instead of grinding through details, the purpose of the course should be to *demonstrate* hardware and software that the audience hasn't seen, *discuss* computer hassles, *let the audience ask lots of questions*, and give the audience *hands-on experience aided by tutors*.

Remember that the schedules given above are just guidelines. Although they guarantee a well-balanced diet of topics, you must tailor the schedules so that they cater to the particular interests of your audience.

If you plan to give such a course and want advice, phone me. I've given many such courses successfully, and I'll do everything possible to help make *your* course a success too.

The first time you give such a course, it might "bomb", because you haven't had enough experience yet in giving demonstrations, fielding questions from the audience, and dramatically varying the pace so that your audience stays awake. The first time you give such a course, play safe: charge as little as possible, so that everybody in the audience feels that the course was a "good deal" and a "wonderful bargain" and nobody feels "ripped off". For your first course, your goal should *not* be money: instead, your goal should be to gain experience and a good reputation.

No matter how great you think you are, your audience will tire of you eventually. To keep your audience awake, give the audience variety, by including your friends as part of your act.

Good luck. Try hard. You can cast a spell over the audience. Courses change lives.

COUPON

SHARE

Share your secret: photocopy the next page, and give it to a friend. If you know a young (or young-at-heart) computerist who's having a birthday, Christmas party, or graduation, turn these volumes into a present: send me the computerist's name, address, and either a greeting card or what you want me to say.

PRICES & POLICIES

The Secret Guide to Computers consists of two volumes. Volume 1 (The Main Secrets) covers everything important about computers; volume 2 (Deep Secrets) digs even deeper. Each volume costs just $14. Do *not* send money for regular (book rate) postage and handling: I'll pay for them myself, even if you live in a foreign country. If you're not sure whether to order a book, go ahead: you can return unused books at any time, for a 100% refund.

DISCOUNTS

The list price is $14 per book; so the two-volume set costs $28. To pay less, get together with a friend:

20% discount if you order 2 sets (or any 4 books); you pay just $11.20 per book
30% discount if you order 4 sets (or any 8 books); you pay just $9.80 per book
40% discount if you order 10 sets (or any 20 books); you pay just $8.40 per book
50% discount if you order 50 sets (or any 100 books); you pay just $7.00 per book
60% discount if you order 500 sets (or 1000 books); you pay just $5.60 per book

START YOUR OWN BOOK BUSINESS

To start your own book business, buy books in quantity from us at discount, and sell them to your friends for slightly under list price. You'll make your friends happy (since they'll pay less than list price), and you'll be happy also (because you'll be helping your friends and also be turning a profit).

IF YOU'RE IN MASSACHUSETTS . . .

If the person who orders the book, or the person we're shipping the book to, is in Massachusetts, we must charge Massachusetts sales tax—unless the book is for a government agency or tax-exempt agency (give us you tax-exempt number) or for resale (give us your Massachusetts resale number) or is a *required text* for a course (tell us which college and course).

PICK-UP

Normally, you get the books by mail. If you live near Boston and don't want to wait for your mailman, come to us and pick up the books yourself: we're near Prudential Center. Before you come, phone us at 617-266-8128, so we can give you directions and a guaranteed pick-up time.

PURCHASE ORDERS

If you work for a government agency (such as a public library or a public school) or an accredited private school, or an established bookstore or computer store, or a major computer manufacturer, we'll accept your purchase order, which you must pay 30 days after receiving the books.

REVIEW COPIES

If you're a reviewer for a large magazine, or the head of a large computer department, you might be able to get a complimentary copy of The Secret Guide to Computers for review. To find out, phone us.

CUT OUT THIS COUPON!

How many copies do you want of volume 1 (The Main Secrets)? ____

How many copies do you want of volume 2 (Deep Secrets)? ____

TOTAL number of books: ____

TOTAL number of books	Price per book, usually	Price per book, if you're a Massachusetts resident (including 5% sales tax)
1 - 3	$14.00	$14.70
4 - 7	$11.20	$11.76
8 - 19	$9.80	$10.29
20 - 99	$8.40	$8.82
100 - 999	$7.00	$7.35
1000 & up	$5.60	$5.88

Multiply your TOTAL number of books by the price per book, and write the TOTAL book price here:

Put a check mark in front of the shipping method you choose. . . .

□ Book rate (or library rate or parcel post or surface), to anywhere in the world: we charge nothing; free!

□ Special handling (which is faster than book rate, and almost as fast as priority mail), to U.S. only: we charge just $1 total, even if your order is large.

□ Priority mail (first-class air-mail): we charge $1.50 *per book* to New England, $2 *per book* to other continental U.S. east of the Mississippi, $2.50 *per book* to other U.S., $6 *per book* to other countries.

FREE brochures on the following topics will become available in April 1984. You can get on our mailing list now, by checking the brochures you want. . . .

□ how you can get later editions of The Secret Guide to Computers

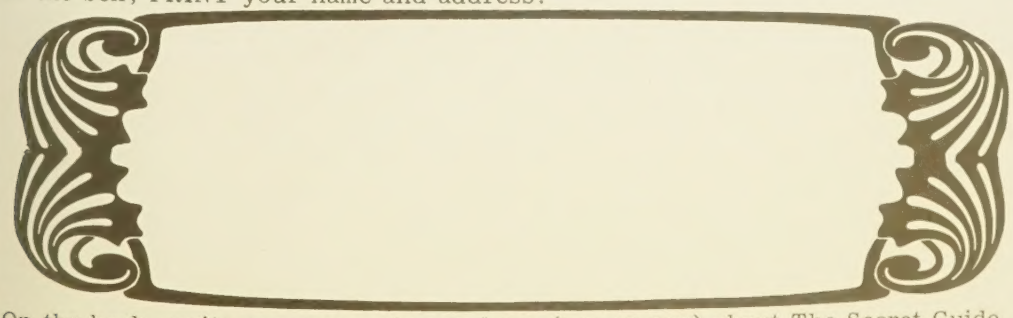
□ how you can get special editions of The Secret Guide to Computers for use by young children and for students in elementary, junior-high, and high school

□ how you can get The Secret Guide to Tricky Living (part 1 is "Sex"; part 2 is "Everything But")

□ how you can attend one-day and two-day courses taught by Russ in various cities

□ how you can attend one-week, four-week, and six-week courses taught by Russ's protégés, at various colleges in the summer

In the box, PRINT your name and address:



On the back, write any comments you have (pro or con) about The Secret Guide to Computers.

Add the total book price to the shipping charges (if any); make the check or money order payable to The Secret Guide to Computers. Send to Russ Walter, 92 Saint Botolph St., Boston, MA 02116.

After riding the waves of the computer revolution,
they all find peace.